Proyecto de Fin de
Master en Ingeniería
de Computadores
Curso 2006-2007

Planificación Simbiótica en Arquitecturas CMP

Autor: Juan Carlos Sáez Alcaide

Director: Manuel Prieto Matías
Departamento de Arquitectura de Computadores y Automática

Facultad de Informática
Universidad Complutense de
Madrid

Índice general

Capítulo 1

Introducción

Este trabajo se centra en el área de diseño de sistemas operativos para multiprocesadores con chips multithreading (CMT). Los procesadores CMT (Chip Multithreading) combinan multiprocesamiento en chip (CMP) con multithreading hardware (MT). Un procesador CMP esta formado por múltiples cores de procesamiento en un solo chip- con uno o más niveles de cache compartidos-, lo cual permite que más de un hilo este activo al mismo tiempo, mejorando la utilización de los recursos del chip. Por otra parte, un procesador MT entrelaza la ejecución de distintos hilos a nivel hardware. De este modo, si un hilo se bloquea por un acceso a memoria o cualquier otra operación costosa en tiempo, otros hilos pueden proseguir su ejecución. En estas arquitecturas, el paralelismo a nivel de hilo (TLP, Thread Level Paralelism) se convierte en paralelismo a nivel de instrucción (ILP, Instruction Level Paralelism) Numerosos estudios han demostrado los beneficios de rendimiento de las arquitecturas CMP y MT [1, 2, 3]

La industria del hardware esta haciendo de CMT su principal mecanismo para mejorar el rendimiento de las futuras aplicaciones, ya que las técnicas convencionales para ocultar la latencia de las operaciones más costosas en tiempo -como la predicción de saltos o la ejecución en desorden- comienzan a no funcionar de manera óptima con las aplicaciones actuales. Este tipo de aplicaciones, tales como web services, servidores de aplicaciones o sistemas on-line de procesamiento de transacciones; están constituidas frecuentemente por múltiples hilos de control que ejecutan breves secuencias de operaciones enteras, y múltiples saltos condicionales. Este esquema de aplicación presenta gran dinamismo, haciendo disminuir la localidad de la memoria caché así como la precisión de la predicción de saltos, provocando numerosas paradas del procesador [4, 5, 6] . A este hecho puede añadirse el creciente hueco entre el

rendimiento del procesador y la latencia de la memoria que provoca un menor aprovechamiento del pipeline del procesador, –según afirman algunos autores, algunos benchmarks presentan un uso del pipeline inferior al 19 % [7]–. Estas afirmaciones indican que la mayor parte del tiempo el pipeline del procesador está infrautilizado.

La principal motivación de las arquitecturas CMT es afrontar estos problemas. Como prueba de ello, la mayoría de los nuevos procesadores comercializados hoy en día por Intel, Sun Microsystems e IBM son MT, CMP o CMT [8, 9, 10]. Los sistemas CMTs pueden estar equipados con decenas de procesadores lógicos[1] como es el caso de Niágara (Sun) que está compuesto por ocho cores (CMP) cada uno con cuatro vías de ejecución mutithreading hardware (MT).

Por otra parte, cabe destacar que las arquitecturas MT y CMP presentan una cantidad notable de recursos compartidos entre los procesadores lógicos, como unidades funcionales o uno o más niveles de caché. Esto provoca que el rendimiento efectivo del sistema se distancie considerablemente del rendimiento ideal. En [11], se afirma que la latencia introducida por sucesivos fallos de cache de L1 puede ocultarse de manera aceptable en arquitecturas de tipo MT, pero una alta contención en L2 puede degradar drásticamente el rendimiento global del sistema (MT y CMP).

Adicionalmente, la mayor parte de los procesadores actuales carecen de mecanismos a nivel hardware para establecer la fracción de cada recurso compartido que se destina a cada procesador lógico[2]. Esto provoca que hilos de distinta prioridad que ejecuten en distintos procesadores lógicos con recursos compartidos, no gocen de un reparto de los recursos acorde a su prioridad. Esta situación hace que los sistemas operativos actuales sobre arquitecturas CMT deban tener en cuenta la compartición de recursos para garantizar la calidad de servicio (QoS), así como lograr que la productividad del sistema no se vea degradada para conseguir tal fin.

Sin embargo, la mayor parte de los sistemas operativos actuales sobre arquitecturas CMP no implementan políticas de calidad de servicio que ofrezcan mecanismos para establecer un reparto de los recursos compartidos en función de la prioridad de los procesos. Este hecho permite la existencia de situaciones en las que procesos menos prioritarios con un uso intensivo de uno o

[1] En este trabajo se utiliza el término procesador lógico como sinonimo de procesador visible a nivel de sistema operativo, en un SO preparado para multiprocesamiento simétrico. Por ejemplo, Niagara de Sun [9] esta compuesto por 32 procesadores lógicos.

[2] Cabe destacar que el IBM Power5 [8] permite asignar diferentes fracciones de recursos compartidos a cada thread. El reparto de recursos se realiza implicitamente asignando distintos ciclos de decodificación a cada thread.

más recursos compartidos –como la cache de segundo nivel– provoquen una disminución considerable del rendimiento de procesos más prioritarios con los que comparten dichos recursos.

Este trabajo presenta un planificador simbiótico a nivel de sistema operativo para arquitecturas CMP que desarrolla una política de calidad de servicio basada en la desactivación de cores. El término simbiosis se utiliza actualmente para referirse a la efectividad con la que se obtiene mayor rendimiento al ejecutar múltiples hilos simultáneamente en arquitecturas multithreading (MT) [12]. Sin embargo, este concepto puede extenderse a arquitecturas CMP (y en consecuencia a arquitecturas CMT) ya que sigue existiendo un notable índice de compartición de recursos (L2 cache o Front Side Bus) cuyo impacto sobre el rendimiento de las aplicaciones actuales sigue siendo crítico [13].

El planificador simbiótico ha sido implementado sobre la versión 2.6.21 de Linux ejecutando sobre una arquitectura CMP de dos vías (Intel Core 2 Duo). En este tipo de arquitecturas, el planificador de Linux 2.6.x garantiza la calidad de servicio para procesos que ejecutan en un mismo core. Sin embargo, el sistema permite la ejecución de dos tareas de distinta prioridad en distintos cores ignorando las posibles degradaciones del rendimiento de la tarea más prioritaria por motivos de conflicto por el uso de los recursos compartidos por los cores. Por este motivo, Linux no ofrece calidad de servicio (QoS) para procesos que ejecuten en distintos cores.

El objetivo del planificador simbiótico es garantizar la calidad de servicio entre ambos cores sin degradar la productividad del sistema. Para desarrollar, una política de calidad de servicio global, el planificador intenta detectar las situaciones en las que procesos menos prioritarios con un uso intensivo de la cache de segundo nivel provoquen una disminución considerable del rendimiento de procesos más prioritarios. Para ello, el sistema monitoriza el comportamiento de ambos procesos durante la ejecución, haciendo uso de los contadores hardware de monitorización del rendimiento (PMCs) del procesador. Cuando el planificador detecta una subida drástica de la tasa de fallos local de L2 del proceso más prioritario en ejecución, procede a la desactivación de un core temporalmente. De este modo, la política de calidad de servicio consigue mejorar el rendimiento del proceso más prioritario realizando desactivaciones eventuales de un core asociadas a la detección de situaciones de conflicto por compartición de recursos.

1.1. Estructura de contenidos

La estructura del documento se describe a continuación:

- Capítulo 2: En este capítulo se realiza un profundo análisis del diseño interno del planificador de Linux mediante el estudio de su estructura y principales tipos de datos, y la descripción de la política de planificación y equilibrado de carga desplegada en entornos multiprocesador.
- Capítulo 3: En él se enumeran las últimas características que han sido incorporadas al kernel Linux para dar soporte específico a las arquitecturas MT, CMP y CMT.
- Capítulo 4: Describe la problemática asociada al diseño de políticas de calidad de servicio sobre arquitecturas CMP así como la situación actual de Linux en este campo.
- Capítulo 5: Este capítulo se destina al análisis del planificador simbiótico diseñado por el autor e implementado sobre la versión de Linux kernel 2.6.21. En la primera parte del capítulo se expone la política de planificación para dar soporte calidad de servicio en arquitecturas CMP de dos vías (dos cores). Los componentes del planificador simbiótico y los detalles de implementación e interacción con el usuario se tratan al final de este capítulo.
- Capítulo 6: Este capítulo muestra el análisis realizado mediante la herramienta de monitorización del rendimiento (incluida en el planificador simbiótico) de un subconjunto de benchmarks de SPEC CPU 2006. El resto de este capítulo se destina al análisis de los resultados de calidad de servicio que muestran las prestaciones del planificador simbiótico.
- Capítulo 7: Analiza los trabajos relacionados
- Capítulo 8: Muestra las conclusiones del proyecto
- Apéndice A: En él se realiza una introducción a los contadores hardware de monitorización del rendimiento describiendo los problemas de diseño asociados con la ejecución especulativa y fuera de orden. Por otra parte, se realiza un estudio de la gestion de los contadores hardware en los procesadores de Intel, concluyendo el capítulo con los modelos de sistema que hacen uso de estos contadores.

Capítulo 2

El planificador de Linux 2.6.21

Linux es un complejo sistema operativo formado por múltiples componentes estructurados en capas y relacionados entre sí. Estos componentes interactúan por medio de operaciones de comunicación y sincronización en un entorno concurrente. El modelo de concurrencia a nivel de sistema operativo presenta múltiples diferencias (como la expropiación a nivel de kernel), con respecto al modelo tradicional de concurrencia de las aplicaciones multithreading a nivel de usuario. Este modelo implica nuevos desafíos que, unidos a la complejidad por la extensión del código del núcleo, dificultan los procesos de desarrollo y depuración unidos a la inclusión de pequeñas modificaciones en el código. Las consecuencias de estas leves modificaciones pueden llegar a ser realmente complejas de predecir debido al fuerte acoplamiento existente entre los distintos componentes del núcleo.

Dentro del núcleo, puede destacarse una parte central que articula los principales mecanismos del sistema, llamados subsistemas del kernel. Estos subsistemas incluyen el gestor de memoria, el sistema de ficheros virtual, el subsistema de comunicación entre procesos, el subsistema de red y el planificador. En la figura 2.1 se muestra la interacción entre los distintos subsistemas así como las capas que los constituyen. Como puede apreciarse, en el centro de todos los subsistemas se encuentra planificador de tareas (scheduler).

La posición central que ocupa el planificador, hace de él un subsistema del que dependan todos los demás. Este hecho tiene las siguientes implicaciones:

- Es crítico en rendimiento: Una pérdida de rendimiento mínima será percibida por todos los subsistemas y repercutirá de forma global en el núcleo implicando una severa degradación del rendimiento y la productividad de todo el sistema.

- Introducir cambios resulta extremadamente complejo: Por la fuerte dependencia que presentan todos los subsistemas de kernel, con el planificador; cualquier modificación efectuada en él afecta indirectamente a todos los demás subsistemas. Esto hace necesario tener en cuenta una gran cantidad de efectos colaterales que podrían llevar a situaciones conflictivas provocadas por dicho cambio.

La misión del planificador de un sistema operativo es decidir cómo, cuándo, y dónde se ejecutarán las tareas que permanecen actualmente en ejecución en el sistema. Esto hace necesario, el mantenimiento de listas de procesos y la clasificación de cada uno estos procesos en base a múltiples características –como la prioridad, la interactividad o la afinidad–; con el objetivo de proceder a un reparto justo del tiempo de CPU ofrecido a cada tarea que a la vez implique mejoras en diversos aspectos -como el porcentaje de uso de CPU, el rendimiento la productividad o la calidad de servicio-.

El planificador, de la versión de Linux 2.6.x, ha sido reescrito por completo, incorporando grandes cambios respecto a la versión 2.4.x. El objetivo del nuevo diseño del planificador ha sido conseguir una notable mejora de sus capacidades, rendimiento y escalabilidad en multiprocesadores [14]. Algunos

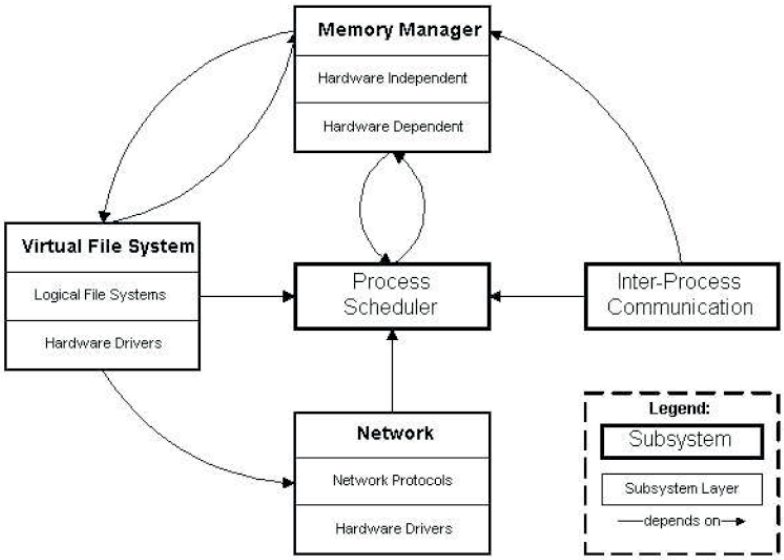

Figura 2.1: Subsistemas de kernel

de los cambios más significativos del planificador de Linux 2.6.x, como la complejidad en $O(1)$ o la expropiación, son expuestos en detalle a lo largo de este capítulo.

2.1. Estructura del Planificador

El planificador de Linux está implementado en kernel/sched.c. El algoritmo fue reescrito por completo en la versión 2.5 –versión de desarrollo- y, al contrario que en las versiones anteriores, fue diseñado para cumplir objetivos específicos:

- Complejidad $O(1)$: Cada algoritmo involucrado en el nuevo planificador debe finalizar en tiempo *constante*, independientemente del número de procesos que haya corriendo en el sistema.
- Escalabilidad *perfecta* en SMP: Cada procesador tiene su propia cola de procesos (runqueue) y su cerrojo asociado de protección contra la concurrencia (lock). De esta manera es posible que de forma simultánea, dos tareas se despierten en distintas CPUs, sean planificadas y se puedan llevar a cabo los cambios de contexto necesarios para que entren en ejecución.
- Afinidad SMP mejorada: El planificador debe intentar agrupar y mantener a los procesos en una CPU específica. Los procesos sólo se migrarán a otra CPU para conseguir una distribución uniforme de la carga entre los procesadores. De este modo, se impide que haya tareas que estén continuamente siendo migradas de una Cola de procesos a otra, y puedan sufrir situaciones de inanición o penalizaciones de rendimiento.
- Eficiencia SMP: Ninguna CPU debe estar inactiva si hay tareas que ejecutar.
- Soporte de planificación por lotes: timeslices largos y política Round-Robin. Se aplica a las tareas con menor prioridad, es decir, con un valor de nice positivo.
- Ejecuta los procesos hijos recién creados antes que sus procesos padres para evitar situaciones de inanición o denegación de servicio.
- Presentan un buen rendimiento con tareas interactivas: Incluso en situaciones con una carga considerable, el sistema debería reaccionar y plani-

ficar las tareas interactivas con una baja latencia –tiempo de respuesta– . Esta característica es extraordinariamente beneficiosa para sistemas tipo desktop.

- Está optimizado para el caso de una o dos tareas en ejecución (baja carga).
- Realiza un reparto justo de los timeslices (*quantos*): ningún proceso debe sufrir inanición ni tener de manera injustificada un timeslice demasiado alto.

Las secciones posteriores analizan detalladamente las principales estructuras de datos involucradas en el planificador.

2.1.1. Procesos

Toda la información relativa a los procesos se guarda en su *descriptor de proceso*. Esta información incluye ficheros abiertos, los datos de su espacio de memoria, señales pendientes de capturar, estado arquitectónico, etc. En Linux, el descriptor de un proceso se representa mediante la estructura struct `task_struct` -definida en include/linux/sched.h-.

A diferencia de otros sistemas operativos, Linux emplea el nombre de tareas (tasks) para referirse aquellas entidades ejecutables con un estado arquitectónico propio (no compartido) que permiten ser planificadas y ejecutadas de manera independiente. El término proceso se utiliza para describir a los programas en ejecución. Un proceso es la entidad mínima de ejecución a nivel de sistema operativo que puede poseer recursos como ficheros abiertos o segmentos de datos independientes en memoria. Sin embargo, un proceso puede estar compuesto por varios hilos de ejecución -cada uno con su propio estado arquitectónico- que puedan comunicarse a través del segmento datos del proceso; compartiendo los recursos que constituyen propiedad del proceso (por ejemplo, descriptores de ficheros abiertos). Sin embargo, los descriptores de proceso tanto de procesos monohilo como de hilos se representan con una estructura del tipo `struct task_struct`. Por tanto el término tarea (task) se utiliza tanto para representar a procesos monohilo como a los distintos hilos de un programa multithreading.

Estructura de los procesos

La estructura struct `task_struct` está compuesta de numerosos campos, pero los más importantes, en lo que a la planificación se refiere, son los

siguientes:

```
struct task_struct {
struct prio_array *array; // Array de prioridades en el que está
struct list_head run_list;// Puntero a la siguiente tarea en el Array de prioridades
int prio;// Prioridad efectiva del proceso
int static_prio;// Prioridad estática del proceso
unsigned int time_slice; // Tiempo de ejecución
unsigned long sleep_avg;// Tiempo medio que pasa suspendido
volatile long state;// Estado del proceso
int activated;// Estado en el que estaba suspendido
unsigned long rt_priority; // Prioridad en tiempo real
unsigned long policy;// Esquema de planificación
cpumask_t cpus_allowed;// CPUs permitidas para este proceso
spinlock_t switch_lock;// Lock para los cambios de contexto.
struct thread_info *thread_info; // Acceso eficiente al Descriptor de Proceso.
struct sched_info sched_info; // Estadísticas a nivel de proceso.
};
```

A continuación se describen algunos de estos campos:

- `struct prio_array *array`: arrays que almacenan a todos los procesos ordenados por su prioridad.
- `struct list_head run_list`: lista de procesos asociada al nivel de prioridad de la tarea.
- `int prio`: es la prioridad efectiva del proceso, tanto para los de tiempo real, como par los procesos de usuario.
- `int static_prio`: Prioridad Estática. Es la prioridad establecida por el usuario para los procesos que no son de tiempo real (procesos de usuario).
- `int timeslice`: Fragmento de ejecución del proceso, dado en ticks de planificador, que ha sido asignado para consumir. Este valor es asignado en función de su prioridad estática.
- `unsigned long sleep_avg`: tiempo medio que el proceso permanece suspendido (sleep average) por E/S. Este valor se obtiene mediante una heurística empleada por el planificador para medir la interactividad de un proceso, y así, ajustar su priori- dad para garantizar un tiempo de respuesta aceptable.
- `volatile long state`: estado del proceso. El significado más detallado de este campo se especifica más adelante.

- `int activated`: este campo es empleado cuando se despierta a un proceso. En función del estado en el que estuvo suspendido, el campo almacena un valor que representa su grado de interactividad. Posteriormente, se modificará su sleep average según este valor.
- `unsigned long rt_priority`: es el nivel de prioridad de los procesos de tiempo real que percibe el usuario
- `unsigned long policy`: es el esquema de planificación empleado para este proceso.
- `cpumask_t cpus_allowed`: máscara que indica los procesadores en los que se puede ejecutar el proceso.
- `spinlock_t switch_lock`: Cerrojo utilizado en algunas arquitecturas durante los para dar soporte a la sincronización necesaria durante los cambios de contexto.
- `struct thread_info *thread_info`: Estructura empleada para acceder de manera eficiente al descriptor de procesos.
- `struct sched_info sched_info`: Estructura utilizada para el almacenamiento de estadísticas a nivel de proceso.

Estados de los procesos

El estado de un proceso se almacena en el campo `state` de la estructura `struct task_struct`. Los estados más importantes en los que puede estar un proceso son los siguientes:

- `TASK_RUNNING`: El proceso esta listo para ejecutar y por tanto se encuentra en el arrayde Prioridad. En este estado el proceso puede estar ejecutando en la CPU asignada,o bien puede estar esperando su turno.
- `TASK_INTERRUPTIBLE`: El proceso está suspendido esperando a que se cumpla alguna condición para poder proseguir con su ejecución. Cuando esto ocurra, el proceso será despertado y puesto en estado `TASK_RUNNING`. Sin embargo, también puede ser despertado de forma prematura a pesar de no haber ocurrido en esta condición, si dicho proceso recibe alguna señal (por ejemplo, SIGTERM). Los procesos interactivos suelen permanecer en este estado cuando están suspendidos.

- `TASK_UNINTERRUPTIBLE`: Este estado es similar al caso anterior, exceptuando que el proceso no será despertado si recibe alguna señal.
- `TASK_TRACED`: Estado para dar soporte a la depuración. El proceso está siendo depurado (breakpoint).
- `TASK_STOPPED`: La ejecución del proceso ha sido detenida de forma externa (por ejemplo, al teclear Control+Z desde el shell durante la ejecución de un proceso en foreground). El proceso queda bloqueado al recibir la señal SIGSTOP, SIGSTP, SIGTTIN o SIGTTOU. Dicho proceso puede volver a un estado ejecutable cuando reciba otras señales durante la depuración.
- `EXIT_ZOMBIE`: El proceso ha finalizado su ejecución, pero su descriptor sigue en memoria para permitir que el proceso creador (padre) pueda acceder a información sobre su ejecución. Dicho descriptor será eliminado cuando el padre ejecute la llamada al sistema wait4() o similares.
- `EXIT_DEAD`: El proceso ha acabado y su descriptor se ha eliminado.

El mecanismo más empleado para cambiar el estado de un proceso es haciendo uso de las macros `set_task_state(task,state)` y `set_current_state(state)`. La diferencia entre ambas es que la segunda sólo cambia el estado del proceso en ejecución en la CPU actual (current). En sistemas SMP, además, puede establecer una barrera para sincronizar a las tareas del resto de procesadores.

Las transiciones entre estados de los procesos, y sus causas, se pueden observar en la figura 2.2.

Accediendo al Descriptor de Proceso

En versiones de Linux anteriores a la 2.6.x, el descriptor de proceso (`struct task_struct`) se almacenaba al final de la pila de de kernel de cada proceso. De este modo, en arquitecturas con pocos registros, como la x86, se empleaba el puntero a la cima de la pila (`esp`) como registro base para acceder a este descriptor.

En esta nueva versión de Linux, los descriptores de procesos son creados de manera dinámica utilizando el `slab_allocator`. Para acceder a ellos, se ha creado una nueva estructura al final de la pila de kernel (donde residía previamente el descriptor de proceso) denominada `thread_info`, en la cual, entre otros campos, hay un puntero al descriptor (campo task).

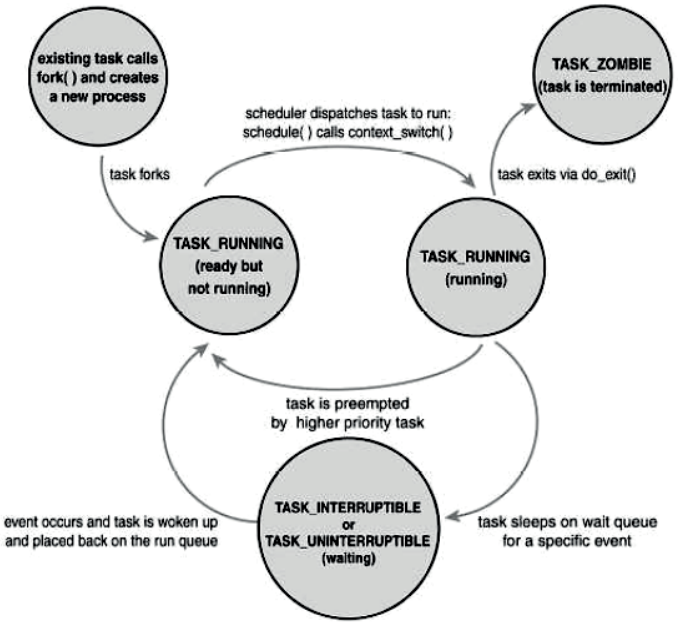

Figura 2.2: Estados de los procesos.

En el código del planificador, es muy común requerir el acceso al descriptor del proceso que está actualmente en ejecución, por ello existe una macro denominada current para acceder a este descriptor de manera más eficiente. Esta macro se implementa de distinta forma para cada arquitectura en include/asm-*/current.h. En algunas de ellas, como la x86, invoca a la función `current_thread_info()` (implementada en ensamblador) para obtener el `thread_info` del proceso, y luego su descriptor. En la figura 2.3 se puede observar la estructura `thread_info` situada en la pila de kernel del proceso.

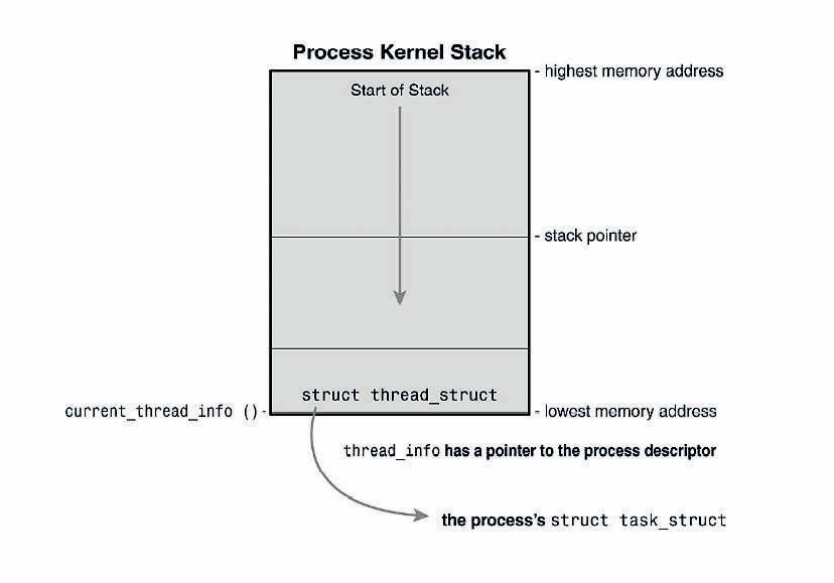

Figura 2.3: Descriptor de proceso en la pila de sistema.

Inicio de un proceso

Tradicionalmente en sistemas Unix, la creación de un proceso con la llamada al sistema `fork()`, implicaba duplicar y copiar casi todos los recursos que poseía el proceso padre en la imagen de memoria del proceso hijo. Con esta aproximación, los procesos padre e hijo sólo se diferenciaban en el PID, el PPID (parent PID) y en algunas estadísticas y recursos tales como las tablas de señales pendientes. Esta implementación resulta bastante ineficiente, debido a que la mayor parte de las creaciones de procesos suelen sucederse con la invocación la llamada a sistema `exec()` por parte del proceso hijo. Esta llamada implica la carga un nuevo programa en el espacio de memoria haciendo que la copia de la imagen de memoria del proceso padre (que fácilmente

puede alcanzar los 10 MB), suponga un derroche de recursos innecesarios.

Actualmente, esta llamada al sistema se implementa empleando un mecanismo denominado *Copy-On-Write* (COW), cuyo objetivo es el de retrasar, o incluso evitar, la copia innecesaria de recursos a la hora de crear un proceso hijo. Con esta técnica, en lugar de proceder a la duplicación de todo el espacio de memoria, el proceso padre y el hijo comparten inicialmente una misma y única copia. De esta forma, la duplicación de los datos que contiene sólo se llevará a cabo si uno de los procesos intenta modificar alguna de las páginas de ese espacio de memoria. Si el proceso hijo realiza una llamada a `exec()` justo después del `fork()`, ninguno de los datos del espacio de memoria del padre habrán sido duplicados. La única carga inevitablemente provocada por `fork()` es la copia de las tablas de páginas del padre y la creación de un nuevo descriptor de proceso para el hijo.

A pesar del uso del mecanismo de COW, aún se pueden llevar a cabo más optimizaciones. El hecho de despertar al proceso hijo antes de continuar con el padre -en previsión de que el hijo ejecutará inmediatamente un `exec()`- evitará realizar un mayor número de duplicaciones innecesarias; ya que, de este modo, se evita que el padre comience a hacer modificaciones en las páginas del espacio de memoria antes de que se invoque un posible `exec()` por parte del proceso hijo. Esta optimización, denominada child-run-first, está incluida en Linux 2.6.x y se puede observar analizando la función `wake_up_new_task()`.

2.1.2. Colas de ejecución

La principal estructura de datos del planificador es la Cola de Ejecución o Runqueue. Una runqueue es una estructura que almacena la lista de procesos ejecutables en cada procesador. Existe una cola de ejecución por procesador y cada proceso en ejecución sólo puede estar presente en una cola de ejecución a la vez. Las colas de ejecución se representan por medio de estructuras de tipo `struct rq`, definidas en kernel/sched.c.

Estructura de la Cola de Procesos

La estructura de la cola de procesos es la siguiente:

```
struct rq {
spinlock_t lock; // Cerrojo para proteger esta cola
unsigned long nr_running; // Número de tareas ejecutables
unsigned long cpu_load; // Carga del procesador
unsigned long long nr_switches; // Número de cambios de contextos
```

```
unsigned long expired_timestamp;// Tiempo desde el último intercambio de arrays
unsigned long nr_uninterruptible;// Número de tareas en estado TASK_UNINTERRUPTIBLE
unsigned long long timestamp_last_tick; // Marca de tiempo del último tick del planificador
struct task_struct *curr; // Proceso en ejecución
struct task_struct *idle; // Proceso ocioso de esta cola
struct mm_struct *prev_mm; // Mapa de memoria virtual del proceso anterior
struct prio_array *active; // Array de prioridad de tareas activas
struct prio_array *expired; // Array de prioridad de taras expiradas
struct prio_array arrays[2]; // Arrays de prioridad
int best_expired_prio; // La mayor de las prioridades de las tareas expiradas
atomic_t nr_iowait; // Número de tareas esperando E/S
struct sched_domain *sd; // Dominio de planificación de esta cola
int active_balance; // Indica si la cola necesita equilibrado
int push_cpu;// CPU a la que se envían tareas durante un equilibrado
struct task_struct *migration_thread; // Proceso para la migración de tareas
struct list_head migration_queue; //Lista de tareas a migrar durante un equilibrado
};
```

A continuación se procede a la descripción de cada campo de la estructura:

- `spinlock_t lock`: cerrojo para proteger el acceso a la cola. De esta manera, sólo una tarea puede modificarla.
- `unsigned long nr_running`: número de procesos ejecutables que están en la cola.
- `unsigned long cpu_load`: carga de trabajo del procesador. Esta carga se calcula y se actualiza en el método `rebalance_tick()` haciendo la media entre la carga anterior y la actual.
- `unsigned long long nr_switches`: número de cambios de contexto que han transcurrido desde la creación de la cola al iniciarse el sistema. Actualmente sólo se utiliza al mostrar estadísticas en el sistema de ficheros /proc.
- `unsigned long expired_timestamp`: tiempo transcurrido desde la última vez que se intercambiaron los arrays de prioridad.
- `unsigned long nr_uninterruptible`: número de tareas que se encuentran en el estado `TASK_UNINTERRUPTIBLE`.
- `unsigned long long timestamp_last_tick`:marca de tiempo del último tick del planificador. Se utiliza principalmente en la macro `task_hot()` para determinar si un proceso es cache hot, es decir, si ha transcurrido muy poco tiempo y por tanto, es muy probable que en la cache de la CPU todavía se encuentren datos válidos del proceso.

- `struct task_struct *curr`: es el proceso que actualmente se está ejecutando en este procesador.
- `struct task_struct *idle:` es el proceso *ocioso* de esta cola, es decir, el proceso que se ejecuta cuando no hay otras tareas a ejecutar.
- `struct mm_struct *prev_mm`: mapa de la Memoria Virtual de la tarea que estuvo ejecutándose anteriormente. Se utiliza para una gestión eficiente de la memoria virtual.
- `struct prio_array *active`: contiene las tareas que no han agotado aún todo su *quanto* (*timeslice*).
- `struct prio_array *expired`: contiene las tareas que ya han agotado todo su *quanto*.
- `struct prio_array arrays[2]`: arrays de prioridad.
- `int best_expired_prio`: la mayor de las prioridades (la del mínimo valor) de las tareas expiradas. Se utiliza en la macro `EXPIRED_STARVING()` para determinar si existe una tarea con mayor prioridad que la actual en el array de tareas expiradas. De esta manera, si la tarea actual es interactiva, no se reinserta en el array de las activas sino en el de las expiradas y asíse evita la inanición de las tareas expiradas.
- `atomic_t nr_iowait`: número de procesos esperando en operaciones de E/S. Utilizado para las estadísticas del kernel.
- `struct sched_domain *sd`: dominio de planificación para esta cola. Básicamente es el grupo de CPUs entre los cuales se hace el equilibrado activo.
- `int active_balance`: flag empleado en la migración de tareas para determinar si la cola debe ser equilibrada, es decir, si está considerablemente más llena que el resto.
- `int push_cpu`: procesador al que son enviadas tareas durante el equilibrado.
- `struct task_struct *migration_thread`: proceso encargado de migrar tareas a otro procesador durante un equilibrado de carga.
- `struct list_head migration_queue`: lista de tareas que deben ser migradas a otra CPU. Dado que las colas de procesos son la principal estructura de datos del planificador, existen una serie de macros

definidas en kernel/sched.c para facilitar la manipulación de las mismas

- `cpu_rq(cpu)`: devuelve un puntero a la cola de procesos del procesador cpu.
- `this_rq()`: devuelve un puntero a la runqueue del procesador actual.
- `task_rq(p)`: devuelve un puntero a la cola donde está el proceso p.

Operaciones de bloqueo

La nueva versión de Linux es *completamente expropiativa*, por lo tanto el planificador puede seleccionar a una nueva tarea para su ejecución tanto si la anterior está ejecutando en modo usuario, como si lo está haciendo en modo núcleo. Esto obliga a proporcionar métodos que controlen el acceso concurrente a los recursos del planificador. Un ejemplo de ello son las runqueues, que poseen mecanismos de bloqueo individuales para permitir modificaciones seguras por funciones del sistema operativo que ejecuten en cualquier procesador (por ejemplo, durante las operaciones de equilibrado de carga). Estos mecanismos de bloqueo se implementan mediante un cerrojo (lock) asociado a cada cola de ejecución que es preciso adquirir antes de llevar a cabo operaciones de lectura/escritura sobre dicha cola. Cabe destacar que dicho cerrojo también ha de ser liberado al finalizar la operación. Estas operaciones de bloqueo y desbloqueo pueden realizarse por medio de las siguientes funciones:

- `task_rq_lock(p, flags)`: Bloquea y devuelve un puntero a la cola en la que el proceso p se está ejecutando. De manera adicional, permite deshabiltar las interrupciones guardando el estado en flags.
- `task_rq_unlock(rq, flags)`: Desbloquea la cola rq y rehabilita las interrupciones con el estado almacenado en flags.
- `this_rq_lock()`: Bloquea y devuelve un puntero a la cola asociada al procesador actual. También deshabilita las interrupciones, pero sin guardar el estado.
- `rq_unlock(rq)`: Desbloquea la cola rq y habilita las interrupciones con el estado por defecto.

Otra posibilidad de llevar a cabo los bloqueos, es acceder directamente al campo `spinlock_t lock` de la estructura `struct rq`, y utilizar los métodos `spin_lock(&rq->lock)` y `spin_unlock(&rq->lock)` para bloquearla y desbloquearla respectivamente. Si además de deshabilitar la expropiación del planificador, también se desea deshabilitar las interrupciones, entonces deben emplearse las funciones `spin_lock_irqsave()` y `spin_unlock_irqrestore()`. En el caso de requerir el bloqueo de varias colas de ejecución, se opta por realizar el acceso en orden ascendente de direcciones para evitar interbloqueos. Esto se puede realizar de manera automática con las funciones `double_rq_lock(rq1, rq2)` y `double_rq_unlock(rq1, rq2)`.

2.1.3. Arrays de prioridad

Los arrays de prioridad son las estructuras de datos claves que han permitido implementar un planificador con coste $O(1)$. Gracias a ellos, se puede conseguir transiciones de época[1] en tiempo *constante*, independientemente del número de procesos en ejecución. Adicionalmente, permiten seleccionar de un modo eficiente, el siguiente proceso a ejecutar.

Cada cola de ejecución almacena dos arrays de prioridad: el de *tareas activas* y el de *tareas expiradas*. Las tareas activas son aquellas que no han consumido todo su *timeslice*, mientras que las expiradas, son aquellas que ya lo han agotado. Cada array de prioridad contiene una lista de procesos por cada nivel de prioridad y un mapa de bits (*bitmap*) de prioridad para encontrar eficientemente la tarea con mayor prioridad (que será la siguiente a ejecutar). Esto se puede ver en la figura 2.1.3. La estructura de estos arrays, definida en kernel/sched.c como struct `prio_array` es la siguiente:

```
struct prio_array {
unsigned int nr_active;             /* Número de procesos en el array. */
unsigned long bitmap[BITMAP_SIZE];  /* Bitmap de prioridad. */
struct list_head queue[MAX_PRIO];   /* Array de colas de procesos. */
};
```

A continuación se explica cada campo:

- `unsigned int nr_active`: es el número de procesos que se encuentran en el array.
- `struct list_head queue[MAX_PRIO]`: es un array de listas enlazadas de procesos. Cada lista se corresponde con un nivel de prioridad (por

[1] Una época es el tiempo transcurrido desde el instante en que todos los procesos obtienen un nuevo *timeslice* (inicio de época) y el momento en el que todos los procesos consumen dicho *timeslice* (fin de época).

defecto, `MAX_PRIO` es 140), de manera que cada proceso se inserta según ese nivel. Por ejemplo, un proceso con prioridad 7 se insertará al final de la lista que hay en `queue[7]`. Por tanto, los procesos de mayor prioridad, serán siempre los de la lista que tenga el índice más bajo en este array. En estas listas, los procesos están enlazados entre sí, a través del campo `run_list` de su estructura `struct task_struct`. (figura 2.5).

- `unsigned long bitmap[BITMAP_SIZE]`: array de `unsigned long` cuyos bits representan los niveles de prioridad de los procesos. `BITMAP_SIZE` es el número de unsigned long necesarios para que haya un total de `MAX_PRIO` bits. Por ejemplo, con 140 niveles de prioridad y variables `unsigned long` de 32 bits, `bitmap[]` debe ser un array de tamaño 5 (160 bits).

Inicialmente, todos los bits almacenan el valor a 0. Cada vez que una tarea se inserta en una de las listas de queue[], el bit correspondiente al nivel de dicha tarea se pone a 1. Por ejemplo, al insertar un proceso con prioridad 7 en queue[7], el séptimo bit de este campo se fija a 1. Para encontrar la tarea de mayor prioridad, basta con buscar el primer bit que valga uno, mediante la función `sched_find_first_bit()` de coste constante. Con este mecanismo se evita tener que recorrer el array de listas, obteniendo así, un algoritmo de complejidad O(1) con en el número de procesos en ejecución.

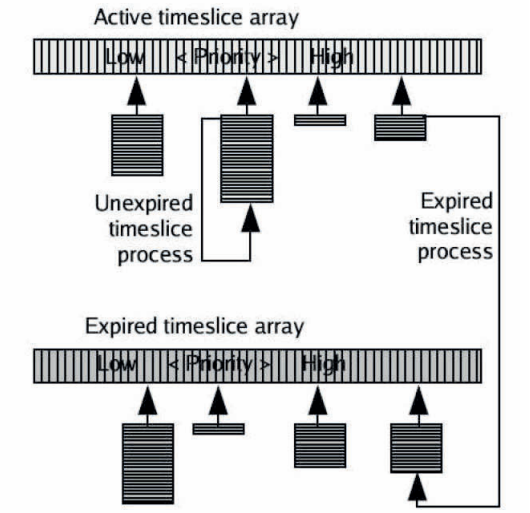

Figura 2.4: Arrays de Prioridad.

El campo prio, de la estructura `struct task_struct` de cada proceso, es el que almacena el nivel de prioridad, y por tanto, es el utilizado como índice para acceder a estos arrays. De hecho, este es el campo utilizado por las funciones `enqueue_task()` y `dequeue_task()`, utilizadas para insertar y eliminar los procesos en los arrays de prioridad. Además, si este proceso es el primero en ser insertado, o el último en ser eliminado, estas funciones también proceden a la activación o desactivación (respectivamente) del correspondiente bit de `bitmap[]`.

Recalculando los timeslices

Muchos sistemas operativos, incluyendo antiguas versiones de Linux, poseen una función empleada explícitamente para recalcular los timeslices de cada proceso al finalizar una época del planificador. Básicamente se trata de un bucle que recorre las listas de tareas y calcula, para cada proceso, un nuevo tiempo de ejecución en función de su prioridad. Este método presenta varias desventajas:

- Es un método de complejidad lineal (O(n)) en el número de procesos en ejecución. Es necesario bloquear la lista de tareas mientras se ejecuta el bucle, lo cual implica que debe detenerse el sistema por completo.
- Dado que dicho recálculo se efectúa de forma asíncrona, hace su utilización inadecuada para procesos de tiempo real.

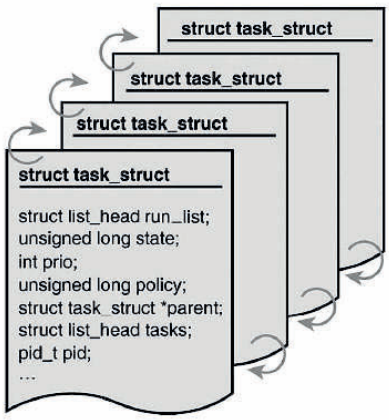

Figura 2.5: Lista de procesos para un nivel de prioridad dado.

El nuevo planificador incluido en la versión 2.6.x, resuelve estos problemas mediante el uso de los arrays de prioridad. Como se mencionó anteriormente, cada Runqueue (es decir, cada procesador) posee dos de estos arrays: el de procesos activos y el de expirados. El de activos contiene a los procesos que no han consumido todo su timeslice, mientras que el de expirados, son todos aquellos que ya lo han agotado.

Cuando una tarea consume todo su timeslice, este es recalculado justo antes de moverla al array de expirados. Cuando el array de activos se queda vacío, entonces se intercambian los punteros que hay en la estructura struct rq. De esta manera, el recorrido del bucle para calcular los nuevos timeslices se reduce a intercambiar dos punteros, acción independiente del número de procesos en ejecución. Estas acciones son llevadas a cabo por las funciones `task_running_tick()` y `schedule()` respectivamente.

2.1.4. Dominios de Planificación

Los dominios de planificación (*scheduling domains*) son conjuntos de procesadores que comparten ciertas propiedades y una misma política de planificación, de manera que pueden repartirse la carga de trabajo. Estos conjuntos son multinivel, ya que se intenta representar la topología hardware del sistema de manera jerárquica.

Este mecanismo surge de la necesidad de proporcionar un sistema de planificación flexible y genérico, que permita gestionar la carga de trabajo en las diferentes topologías de los sistemas multiprocesadores actuales. En los distintos niveles de esta jerarquía, los procesadores se relacionan entre sí de manera distinta.

Por ejemplo, en una CPU SMT (mutithreading simultáneo), los procesadores lógicos comparten la memoria principal, las memorias caches e incluso las unidades funcionales. En este caso, no existe afinidad a la memoria cache (cache affinity), de manera que cualquier proceso que estuvo ejecutándose en uno de estos procesadores y fue suspendido, puede ser reactivado en otro sin perder los datos que tenía en esa memoria, y por tanto, sin generar fallos de acceso. El equilibrado de carga puede realizarse con bastante frecuencia en este caso.

Por otro lado, en los sistemas SMP (*Symmetric Multi-Processing)* sólo se comparte la memoria principal y cada procesador posee sus propios recursos internos. Aquí sí existe afinidad a la cache. Por lo tanto en SMPs no es conveniente realizar un equilibrado de carga con demasiada frecuencia.

Como situación intermedia a las dos anteriores, puede destacarse el caso particular de compartición existente entre los procesadores lógicos o cores de los sistemas CMP (Chip multi-processing). En estos sistemas el chip está formado por varios cores que comparten ciertos recursos como algún nivel de cache o el front side bus. Habitualmente, las implementaciones actuales constan de varios cores -cada uno de ellos con su cache independiente de primer nivel- en las que cada uno de ellos hace uso de una cache de segundo nivel compartida (L2). En esta situación si existe afinidad a la cache de primer nivel, pero no a la cache de L2. Una migración de una tarea de un core a otro dentro del mismo chip, provocará que la tarea pierda la afinidad a la cache de primer nivel, pero mantendrá los datos de la cache de L2.

Teniendo en cuenta un sistema SMP con CPUs SMT, una política de equilibrado que trate a todos los procesadores lógicos por igual durante un equilibrado de carga, no da lugar a la obtención de los mejores resultados. En el caso de que el sistema tuviera únicamente dos tareas en ejecución completamente independientes, ubicar ambas tareas en distintas CPUs físicas constituirá una mejor solución que hacerlo en procesadores lógicos de la misma CPU SMT. Si las CPUs del sistema SMP fueran CMPs multicore, la decisión de ubicar las tareas en cores del mismo o de distinto paquete físico dependerá de los objetivos de la política de equilibrado. La ubicación de ambas tareas en cores de un mismo chip, permitirá -por lo general- obtener un mejor consumo que la solución cuyo objetivo es optimizar el rendimiento -ubicando ambas tareas en distintos chips-.

Finalmente, en los sistemas NUMA (*Non-Uniform Memory Access*), cada nodo puede tener una latencia de acceso diferente para distintas posiciones de la memoria principal. La afinidad a la cache es bastante duradera y, por consiguiente, sólo se debería migrar un proceso de un nodo a otro de manera esporádica, ya que puede llegar a constituir una operación bastante costosa.

En consecuencia, el principal problema al que el planificador debe enfrentarse en sistemas con más de un procesador, es el equilibrado de carga, ya que no es deseable tener unos procesadores con mucha carga de trabajo mientras que otros están ociosos. Por otra parte, la migración de tareas entre procesadores es una operación costosa y debe hacerse de forma justificada.

Estructura de los Dominios de Planificación

Existen dos tipos de estructuras básicas involucradas en el equilibrado de carga. La primera es `struct sched_domain`, donde se almacena toda la información necesaria para el equilibrado de carga. La segunda es la estructura

`struct sched_group` usada para representar grupos de procesadores que son tratados como una unidad durante dicho equilibrado, es decir pueden ser origen o destino de una migración de tareas. A continuación se introducen sucesivos ejemplos explicativos.

Se analiza el caso de un sistema SMP con dos CPUs SMT y que cada una de ellas, contiene dos procesadores lógicos (o virtuales). Con los dominios de planificación, cada CPU física representaría un dominio con una estructura `struct sched_domain`, la cual incluiría sus dos respectivos procesadores lógicos, cada uno en un grupo `struct sched_group`. Estos dos dominios tendrían un dominio padre común, el cual contendría a los cuatro procesadores lógicos agrupados en dos estructuras `struct sched_group`. Esta jerarquía se representa en la figura 2.6.

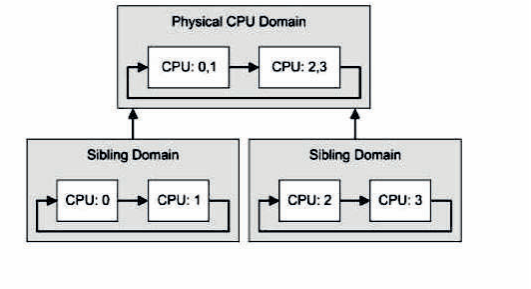

Figura 2.6: *Scheduling Domains* en un sistema SMP con procesadores SMT.

Ahora consideremos un sistema NUMA con dos nodos, cada uno de los cuales es una máquina SMP que contiene dos CPUs no SMT (sin procesadores lógicos). En este caso, cada sistema SMP representa un dominio, que incluye a sus respectivas CPUs en dos grupos. Al igual que en el caso anterior, existe un dominio padre común al de ambos nodos, el cual contiene dos grupos con las cuatro CPUs. Esta nueva jerarquía queda ilustrada en la figura 2.7.

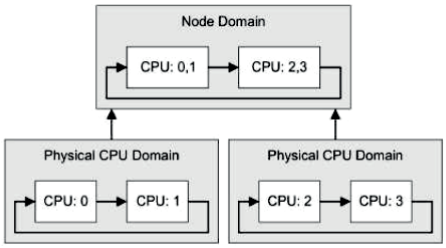

Figura 2.7: *Scheduling Domains* en un sistema NUMA con nodos SMP.

Finalmente, un sistema que fuese la combinación de los dos anteriores, presentaría una jerarquía de tres niveles: cuatro dominios SMT agrupados en dos SMP, los cuales tienen un dominio padre común. Esta situación está representada en la figura 2.1.4.

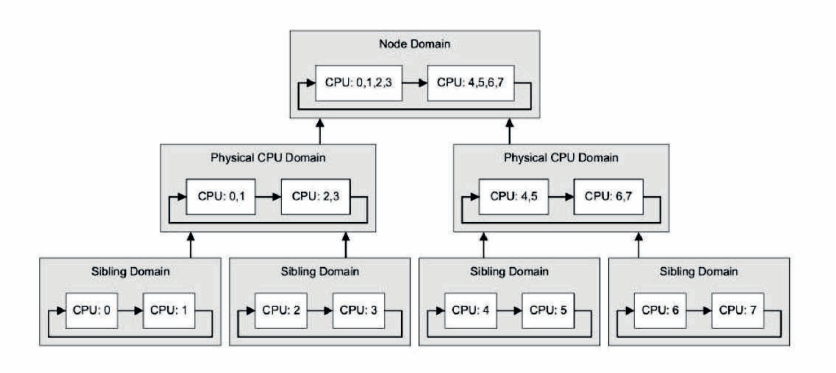

Figura 2.8: *Scheduling Domains* en un sistema NUMA con nodos SMP y procesadores SMT.

Implementación de los Dominios de Planificación

La implementación de la estructura de los dominios de planificación, y sus campos más importantes, son los siguientes:

```
struct sched_domain {
struct sched_domain *parent; // Dominio de planificación padre
struct sched_group *groups; // Grupos de equilibrado del dominio
cpumask_t span; // Procesadores pertenecientes al dominio
```

```
unsigned long min_interval; // Mínimo intervalo de equilibrado (ms)
unsigned long max_interval; //Máximo intervalo de equilibrado (ms)
unsigned int busy_factor; //Factor de reducción del equilibrado si hay mucha carga
unsigned int imbalance_pct; // Umbral de desequilibrado
unsigned long long cache_hot_time; // Tiempo de validez de los datos en la cache (ns)
unsigned int cache_nice_tries; // Número de veces que se dejan los procesos cache hot
unsigned int per_cpu_gain; // Ganancia al añadir dominios de CPU
int flags; // Flags de los dominios de planificación
unsigned long last_balance; //  equilibrado Ultimo (jiffies)
unsigned int balance_interval; // Intervalo entre equilibrados (ms)
unsigned int nr_balance_failed; // Número de equilibrados fallidos.
};

struct sched_group {
struct sched_group *next; // Puntero a otro grupo de procesadores
cpumask_t cpumask; // Procesadores del grupo
unsigned long cpu_power; // Capacidad de computación del grupo.
};
```

Por razones de eficiencia, la estructura `struct sched_domain` no se comparte entre los procesadores que pertenecen al dominio que esta representa, sino que cada procesador tiene su propia copia. Sin embargo, las estructuras de los grupos de procesadores (`struct sched_group`) son compartidas por todos los procesadores. Por ejemplo, para el sistema SMP con dos CPUs SMT (cada una con dos procesadores lógicos), explicado anteriormente, el árbol de dominios sería como el que se muestra en la figura 2.9.

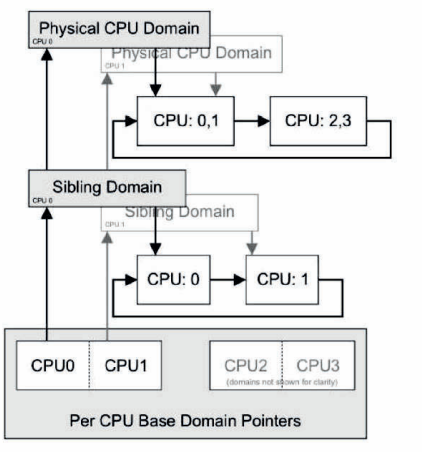

Figura 2.9: Implementación de los *Scheduling Domains* en un sistema SMP con procesadores SMT.

Durante el proceso de inicizalización del núcleo, los campos de la estructura `struct sched_domain` toman valores por defecto según la topología del sistema. Estos valores inciales se encuentran definidos en `include/linux/topology.h`, salvo los de los sistemas NUMA, que deben ser definidos específicamente para cada arquitectura en `include/asm-*/topology.h`.

Política de equilibrado de carga

La política de equilibrado de carga se define en cada dominio con una combinación de valores en el campo `flags` de la estructura `struct sched_domain`. Todos los posibles valores de este campo se muestran en la tabla 2.1.

Flags	Descripción
`SD_LOAD_BALANCE`	Habilita el equilibrado de carga en el dominio
`SD_BALANCE_NEWIDLE`	Hace equilibrado cuando quede poca carga de trabajo
`SD_BALANCE_EXEC`	Realiza equilibrado en las llamadas a `exec()`
`SD_WAKE_IDLE`	Despertar los procesos en las CPUs del dominio que estén ociosas
`SD_WAKE_AFFINE`	Permite que los procesos sean despertados en otras CPUs del dominio
`SD_WAKE_BALANCE`	Equilibrar cuando se despierte un proceso
`SD_SHARE_CPUPOWER`	Miembros del dominio comparten los recursos de la CPU

Cuadro 2.1: Flags en los Scheduling Domains

Tanto en sistemas SMP como SMT, se establecen por defecto los flags `SD_LOAD_BALANCE`, `SD_WAKE_IDLE`, `SD_WAKE_AFFINE` y `SD_BALANCE_NEWIDLE`, ya que las penalizaciones por mover procesos a otras CPUs del mismo dominio, son escasas (o nulas, como en el caso de arquitecturas SMT). Además, también se activa el flag `SD_BALANCE_EXEC`, ya que en una llamada a `exec()`, el proceso pierde toda afinidad con el procesador (ya no necesita los datos que haya en su memoria cache), y por tanto es el mejor candidato para ser cambiado de CPU durante un equilibrado de carga.

Sin embargo, el flag `SD_SHARE_CPUPOWER` sólo se establece en arquitecturas SMT para indicar que todos los procesadores del mismo dominio comparten los recursos de la CPU física, y por tanto, hay que dejarlos libres para las tareas más prioritarias.

Del mismo modo, el flag `SD_WAKE_BALANCE` solamente se activa en sistemas SMP para permitir que un proceso sea despertado en otra CPU, del mismo dominio, si existe un desequilibrio en la carga de trabajo. Esto es así, porque en los sistemas SMP, los equilibrados de carga son menos frecuentes que en los sistemas SMT.

Finalmente, en los sistemas NUMA, las arquitecturas que definen valores

iniciales no suelen activar los flags de equilibrado de carga en el nivel más alto de la jerarquía, ya que mover procesos de un nodo a otro, constituye una operación costosa que debe realizarse esporádicamente.

2.2. La función de planificación

La función principal del planificador es `schedule()`. Su propósito es seleccionar el siguiente proceso a ejecutar. Esta función es invocada cuando un proceso desea ceder el procesador (a través de la llamada al sistema `sched_yield()`), o cuando este debe ser expropiado.

Podemos dividir los pasos llevados a cabo por esta función en cuatro secciones o partes:

1. Acciones iniciales: Durante el primer paso es preciso es realizar una serie de preparativos y comprobaciones, tales como:
   - Verificar que la función no fue invocada mientras el kernel está en modo atómico o con las interrupciones deshabilitadas, ya que podría provocar interbloqueos.
   - Verificar que el proceso en ejecución no es la tarea ociosa (aquella que se ejecuta cuando no hay otros procesos a ejecutar) y que esta no está en estado `TASK_RUNNING`.
   - Deshabilitar la expropiación y determinar el tiempo que el proceso actual ha estado ejecutándose. Si el proceso era interactivo, dicho tiempo es reducido para evitar que este tipo de procesos –que suelen estar esperando en operaciones de E/S– pierdan su estatus de interactivo debido a que puntualmente han estado ejecutándose durante un largo período de tiempo.
   - Si se está realizando una expropiación, las tareas en estado `TASK_ _INTERRUPTIBLE` que tengan alguna señal de interrupción pendiente pasan a estado `TASK_RUNNING`, mientras que las que están en estado `TASK_ UNINTERRUPTIBLE` se eliminan de la lista de tareas. Este procesamiento es necesario, ya que los procesos interrumpibles con una señal pendiente, necesitan tratarla, mientras que los ininterrumpibles no deberían estar en la cola de procesos.
2. Búsqueda de procesos candidatos: Durante el procesamiento se verifica la existencia de tareas listas para ejecutar. Esta acción se realiza mediante los siguientes pasos:

- Si no hay procesos listos para ejecutar en la cola, se procede a un equilibrado de carga. Si aun así, no hay tareas, entonces se elige a la tarea ociosa (idle task).
- Si por el contrario, había tareas para ejecutar, primero se verifica si puede seleccionarse una de ellas o deben dejarse suspendidas. Esto se realiza mediante la función `dependent_sleeper()`, la cual también pertenece a la planificación SMT, y que por tanto, solo tendrá efectos si está instalado dicho tipo de planificación.
- Si llegado a este punto, ya no quedan procesos en el array de tareas activas, se hace un intercambio entre los punteros de los dos arrays. El procedimiento del kernel 2.4 de esta característica se realizaba mediante un bucle de recálculo de timeslices con complejidad lineal con respecto al número de procesos de la cola de ejecución.

3. Selección de la tarea a ejecutar: Una vez se ha llegado a este paso, se garantiza la existencia de tareas listas para ejecutar en el array de procesos activos. Para seleccionar la siguiente tarea a ejecutar se realizan los siguiente pasos:

   - Se invoca a la función `sched_find_first_bit()` para que localice de manera eficiente, en el campo `bitmap[]` del array, el nivel de prioridad más alto donde haya alguna tarea ejecutable. Con esto, se evita tener que efectuar un recorrido lineal en el array para localizar dicha tarea.
   - El planificador, selecciona la primera tarea de la lista correspondiente al nivel de prioridad encontrado. Por ejemplo, si la función `sched_find_first_bit()` devuelve 7, entonces el proceso de mayor prioridad es el primero de la lista de tareas que se encuentra en `array->queue[7]`. Esta situación puede apreciarse con más claridad en el ejemplo mostrado en la figura 2.10 y en la descripción de los campos de los Arrays de prioridad.
   - Si el proceso seleccionado no es de tiempo real, estaba anteriormente suspendido y en estado distinto a `TASK_UNINTERRUPTIBLE`, entonces es muy probable que se trate de un proceso interactivo. En este caso es preciso actualizar la prioridad dinámica de la tarea invocando a la función `recalc_task_prio()`. Finalmente se actualiza su posición en el array de prioridad.

4. Cambio de contexto: Una vez seleccionado el siguiente proceso a ejecutar, se procede al cambio de contexto entre este último y el proceso actual (current) de la siguiente manera:

- Desactiva el flag `TIF_NEED_RESCHED` en el proceso que va a ser expropiado.
- Actualiza el tiempo que el proceso ha permanecido en ejecución desde el ultimo cambio de contexto o `scheduler_tick()` (interrupción de reloj del sistema)
- Actualiza las marcas de tiempo del proceso.
- Realiza el cambio de contexto
- Rehabilita la expropiación, que había sido desactivada durante el proceso de selección de la siguiente tarea a ejecutar. Si durante el tiempo transcurrido durante todo el procesamiento, se ha producido alguna solicitud de expropiación, el algoritmo vuelve a comenzar para escoger una nueva tarea mediante `schedule()`.

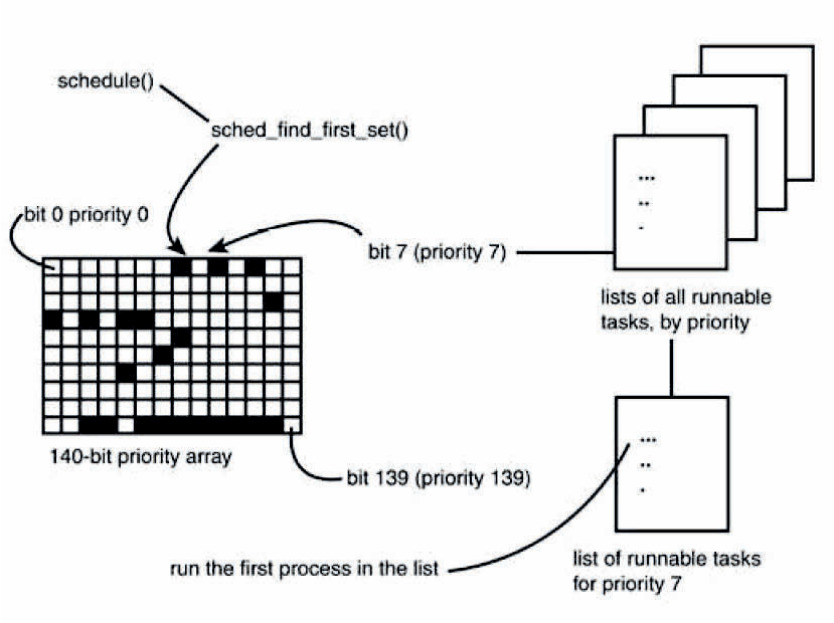

Figura 2.10: Algoritmo de planificación en Linux 2.6.X.

2.3. Equilibrado de carga

Por motivos de escalabilidad, el mecanismo de planificación de tareas empleado por Linux en entornos multiprocesador no se realiza de forma global

–utilizando una lista con todos los procesos del sistema– sino gestionado colas de procesos independientes por procesador. De este modo, la planificación para cada procesador se realiza de manera distribuida. Sin embargo, es preciso disponer de un mecanismo que permita implementar, en último término, una política de planificación a nivel global. Por esta razón, Linux introduce las políticas de equilibrado de carga.

Un desequilibrio en la carga puede producirse como consecuencia de la creación, la terminación o el bloqueo de algún proceso. El procesamiento para detectar la existencia de un desequilibrio implica realizar bloqueos de las colas de ejecución para consultar la carga de las mismas de una manera segura. El chequeo de un posible desequilibrio no puede realizarse con excesiva frecuencia ya que puede llegar a introducir una notable sobrecarga en el sistema.

2.3.1. Equilibrado activo y pasivo

Mientras que algunos eventos –posibles fuentes de desequilibrios de carga– ocurren muy frecuentemente (como el bloqueo por E/S de un proceso); otras situaciones que también pueden implicar desequilibrios, surgen con menos frecuencia (como la creación de un nuevo proceso). Esta situación, unida a la sobrecarga que implica la realización de una comprobación de desequilibrio, motiva la introducción de dos tipos de equilibrado de carga en Linux: equilibrado activo y equilibrado pasivo.

El equilibrado de carga pasivo es el proceso de equilibrado desencadenado por un evento de desequilibrio poco frecuente (como la creación o la destrucción de un proceso). Por otra parte, el equilibrado de carga activo asegura que las colas de ejecución permanezcan equilibradas -ante la aparición de eventos de desequilibrio más frecuentes-, efectuando comprobaciones de equilibrado de carga cada cierto tiempo. Con estos dos tipos de equilibrado, se garantiza un equilibrio de la carga entre todas las colas de ejecución sin introducir sobrecargas adicionales en el sistema.

Equilibrado pasivo

El equilibrado de carga pasivo se lleva a cabo cuando se registran los siguientes eventos:

1. Una cola de ejecución se queda vacía: En esta situación, el planificador invoca la función `idle_balance()`, pasando como parámetro la CPU

y la cola de ejecución asociada que va a quedarse vacía. Las principales funciones involucradas en el procesamiento son:

- `load_balance_newidle()`: función que busca la cola de ejecución más sobrecargada del grupo más ocupado de un dominio de planificación.
- `move_tasks()`: función que intenta conseguir el equilibrio entre dos colas de ejecución migrando las tareas de una cola de ejecución a otra (en este caso la CPU que queda ociosa).
- `can_migrate_task()`: función que permite saber si una tarea es candidata a ser migrada. Esta función rechaza para una posible migración las tareas que:
  a) Están en ejecución.
  b) No son afines a la CPU de destino
  c) Son *cache-hot* en su CPU, es decir tareas que han abandonado muy recientemente dicha CPU y por lo tanto es muy probable que se mantengan en cache datos e instrucciones de dicho proceso.

  Teniendo en cuenta estos casos, se evita llevar a cabo una migración que conlleve un impacto negativo en el rendimiento.

2. Un proceso invoca la llamada al sistema execve(): La región de código de un proceso, ha de recargarse cuando dicho proceso realiza una llamada execve(). Este hecho provoca que el proceso pierda toda la afinidad con la cache, constituyendo un perfecto candidato para llevar a cabo una migración. Desde la función `sched_exec()` -invocada por la llamada al sistema que implementa `execve()`), se busca la CPU más ociosa de un dominio de planificación -al que pertenezca la CPU- que tenga el flag `SD_BALANCE_EXEC` activo. Una vez encontrada, la tarea que invocó el execve es migrada a esa CPU. El procedimiento de migración en este caso es llevado a cabo por los hilos de migración.

3. Creación de un nuevo proceso: Esta situación se produce cuando un proceso invoca la llamada al sistema fork() para la creación de otro. La existencia de un nuevo procedo puede dar lugar a una situación de desequilibrio en la carga.

Equilibrado activo

El procesamiento asociado al equilibrado de carga activo se lleva a cabo cada cierto tiempo para garantizar el equilibrio de la carga de entre los grupos de procesadores (`sched_groups`) pertenecientes a un mismo dominio de planificación. El intervalo de tiempo que transcurre entre cada equilibrado activo depende de la carga de cada CPU. Cuanto más homogénea sea la carga, más largo es el periodo de tiempo transcurrido entre dos equilibrados activos ejecutados de forma consecutiva.

El periodo de equilibrado activo es único para cada dominio de planificación. El valor de este periodo, dado en milisegundos, se almacena en el campo `balance_interval` de la estructura `struct sched_domain`. Adicionalmente, es necesario garantizar –por motivos de sobrecarga– que los procedimientos de equilibrado no se emitan de forma simultánea en cada CPU. Finalmente, para llevar un control absoluto de los tiempos de equilibrado, es preciso conocer el tiempo en el que se realizó el último equilibrado. Dicho valor se almacena en el campo `last_balance` de la estructura `sched_domain`.

La función `scheduler_tick()` –invocada cada tick de planificador– gestiona el control de los intervalos de activación del equilibrado de carga activo. Cuando es preciso llevar a cabo un equilibrado de carga activo, `scheduler_tick()` lanza una interrupción. La rutina de tratamiento de la interrupción (ISR) es la función `run_rebalance_domains()` que comprueba la existencia de equilibrio en la carga, a todos los niveles de la topología de la máquina (dominios de planificación). En el caso de que un dominio de planificación presente desequilibrios, entonces se procede a invocar aquellas acciones necesarias para llegar a una situación de equilibrio entre los grupos de procesadores pertenecientes a dicho dominio (función `load_balance()`).

El procesamiento que lleva a cabo la función `load_balance()` es el siguiente:

- Comprueba que la CPU actual está equilibrada dentro del dominio pasado como parámetro a la función.
- En caso contrario intenta mover tareas a la CPU hasta alcanzar una situación de equilibrio. Para ello obtiene la cola de ejecución con mayor número de procesos dentro del grupo más ocupado del dominio de planificación. El procedimiento de migración de tareas se efectúa mediante la función `move_tasks()`.

En algunos casos pueden presentarse situaciones en las que la función `load_balance()` no consiga garantizar un equilibrio dentro del dominio de planificación. Estas situaciones se producen cuando la función `move_tasks()` no

es capaz de llevar a cabo un equilibrado, por la ausencia de procesos candidatos para una migración. En este caso, el procedimiento de equilibrado se lleva a cabo mediante peticiones efectuadas a los hilos de migración.

Kernel Threads

Antes de introducir el concepto de *hilos de migración*, es preciso introducir los *hilos de kernel* (*kernel threads*). A menudo, resulta útil dotar al kernel de la capacidad de ejecución de ciertas operaciones en segundo plano. Linux lleva a cabo este tipo de operaciones a través de procesos cuya ejecución se restringe a modo núcleo, los hilos de kernel.

La principal diferencia entre los hilos de kernel y los procesos normales es que los primeros no tienen un espacio de direcciones asociado. De hecho, el campo mm de su descriptor de proceso (`struct task_struct`) es nulo. De manera adicional, los hilos de kernel no cambian de contexto a modo usuario, ya que su ejecución se restringe a la ejecución de una función definida en el código del núcleo. Sin embargo, los kernel threads, al igual que el resto de procesos, también son planificables y pueden ser expropiados.

Linux delega algunas tareas a los hilos de kernel, como la tarea *pdflush* y la tarea *ksoftirqd*. La creación de estos hilos se lleva a cabo por parte de otros kernel threads durante el proceso de arranque del sistema. Este aspecto muestra otra propiedad de los kernel threads: la capacidad de ser creados desde otro hilo de núcleo.

La función que permite la creación de un nuevo hilo de kernel desde otro existente es:

```
int kernel_thread(int (*fn)(void *), void * arg, unsigned long flags)
```

La nueva tarea es creada del mismo modo que los procesos de usuario, haciendo uso de la llamada al sistema `clone()` –invocada por la funcion `kernel_thread()`–. Cuando dicha función finaliza, el hilo de kernel padre recibe un puntero al descriptor de proceso (`struct task_struct`) del hilo creado. El proceso hijo ejecuta la función dada por fn, invocada con el argumento arg.

La principal diferencia en la invocación de `clone()` se presenta en el argumento flags pasado a clone() argument. El flag especial de `clone()`, `CLONE_KERNEL`, habilita un conjunto de flags característicos de los kernel threads: `CLONE_FS`, `CLONE_FILES`, and `CLONE_SIGHAND`. La mayor parte de los hilos de kernel son creados activando estos flags en el argumento de clone().

Frecuentemente, un kernel thread permanece en el sistema hasta que la máquina se apaga o se reinicia. De este modo, la función principal del hilo se implementa tradicionalmente como un bucle infinito que sigue este esquema:

- El hilo se despierta ante una petición explícita
- Ejecuta las acciones pertinentes
- El hilo se pone a dormir, esperando sucesivas peticiones

Hilos de migración

Los *hilos de migración* son kernel threads de tiempo real, asociados a cada CPU, cuya funcionalidad es efectuar la migración de tareas de su CPU a cualquier otra CPU destino. Su procedimiento de inicialización se lleva a cabo en la función `migration_call()` –función invocada tras la inicialización de cada CPU–.

Estos característicos hilos de núcleo, permanecen dormidos esperando solicitudes de migración de tareas. Pueden ser despertados explícitamente para ejecutar equilibrados activos seguros o bien, para procesar listas de migración de tareas. Se dice que los equilibrados activos de los hilos de migración son seguros, ya que son capaces de migrar cualquier tarea de su CPU. Su naturaleza de kernel threads de tiempo real, permite que los hilos de migración puedan expropiar a cualquier tarea de usuario de la CPU y proceder a su migración a otra CPU destino.

Cada hilo de migración dispone de una cola de peticiones de migración. Para solicitar una migración, es preciso insertar las peticiones en dicha cola. La cola de peticiones de migración se implementa como una lista enlazada, y se almacena en el campo `migration_queue` de la cola de ejecución (runqueue) del procesador donde ejecuta el hilo de migración. Dicha lista enlazada, está compuesta por estructuras enlazadas del tipo `struct migration_req` que permiten especificar una solicitud de migración.

Dichas estructuras contienen los siguientes campos:

- `list`: la cola de migración a la que se ha añadido.
- `task`: puntero a la tarea que se quiere migrar.
- `dest_cpu:` CPU a la que se desea mover la tarea.

- `done`: variable de condición. Se utiliza para efectuar las operaciones de sincronización entre el hilo de migración y el planificador. Al finalizar su trabajo el hilo invoca la función `complete()` sobre la variable condición. En ese preciso instante, todos los procesos que estén esperando en la función `wait_for_completion()` invocada con esa variable condición se desbloquean.

El hilo de migración utiliza la función `__migrate_task()` para mover la tarea de una CPU a otra. Esta función bloquea las dos colas involucradas en la migración antes de proceder con el movimiento. Si esa tarea expropia por prioridad a la que esté corriendo en la CPU destino, se lleva a cabo una replanificación para ajustar las prioridades de las tareas.

2.4. Política de planificación

La política de planificación es la responsable de un aprovechamiento óptimo del procesador, ya que permite determinar qué proceso debe ejecutarse en cada momento. De manera adicional, la selección del siguiente proceso a ejecutar debe garantizar un equilibrio entre diversos factores: latencia (o tiempo de respuesta), rendimiento y productividad.

Para cumplir estos objetivos, se procede a la diferenciación entre dos tipos de procesos: los I/O-bound y los CPU-bound. Los primeros son aquellos que pasan la mayor parte del tiempo suspendidos esperando a que finalicen operaciones de E/S, mientras que los segundos están continuamente utilizando los recursos del procesador. Los procesos I/O- bound sólo permanecen en ejecución durante cortos periodos de tiempo, que desembocarán en una situación de suspensión del proceso por operaciones de E/S. Por otra parte, las tareas CPU-bound pueden permanecer en ejecución hasta que sean expropiados, expulsadas de la CPU de manera involuntaria.

Con el objeto de garantizar una baja latencia, los procesos interactivos (I/O-bound) han de gozar de una mayor prioridad dinámica que los procesos intensivos en CPU para poder entrar a ejecución en el momento en que están disponibles. De esta manera, se consigue que los procesos interactivos comiencen sus peticiones de E/S lo antes posible, dejando libre el procesador para los procesos que más lo necesitan: los CPU-bound. El cálculo de prioridades y de periodos de ejecución (timeslices) se explica con detalle en las siguientes secciones.

2.4.1. Prioridades de los procesos

Linux emplea una planificación basada en prioridades, que realiza clasificaciones de los procesos según la prioridad asignada por el usuario (prioridad estática) y según su comportamiento durante la ejecución (prioridad dinámica). De esta manera, los procesos con mayor prioridad efectiva –combinación de prioridades estática y dinámica– siempre entrarán a ejecutar antes que los de inferior prioridad. Si existen varios procesos con misma prioridad, estos procederán a ejecutarse por medio de turnos rotatorios (round- robin).

Las prioridades en Linux se representan internamente por valores pertenecientes al rango `[0..MAX_PRIO-1]`. Por defecto la constante `MAX_PRIO` (definida en include/linux/sched.h) tiene un valor de 140. Ese valor define el número de niveles de prioridad que existen en el sistema. Cuanto menor sea este nivel para un proceso, mayor será su prioridad.

Estos 140 niveles se encuentran divididos en dos subrangos. El primero de ellos está reservado para procesos de tiempo real y comprende a los primeros `MAX_RT_PRIO` niveles; es decir, el rango `[0..MAX_RT_PRIO-1]`, donde la constante `MAX_RT_PRIO` (también definida en include/linux/sched.h) tiene un valor de 100. El segundo subrango (`[MAX_RT_PRIO..MAX_PRIO-1]`) –tratado con detalle a continuación–, incluye a los 40 niveles restantes, reservados tradicionalmente para los procesos de usuario.

Prioridades Estáticas

Durante la creación de un proceso de usuario, se procede la asignación de una prioridad inicial –valor de prioridad poseído actualmente por el proceso padre–. Esta prioridad se denomina prioridad estática, ya permanece inalterable durante todo el ciclo de vida del proceso. El nivel de prioridad al que pertenece un proceso se almacena en el campo `static_prio` de la estructura `struct task_struct`.

Para llevar a cabo la modificación de la prioridad estática de un proceso en tiempo de ejecución, dicho proceso debe invocar la llamada al sistema `nice()`. Durante su procesamiento, el sistema invoca la función `set_user_nice()` -en kernel/sched.c- que establece la prioridad estática del proceso y actualiza su posición en el respectivo array de prioridades. Sin embargo, en esta llamada al sistema –`sys_nice()`–, el valor de la nueva prioridad introducido como parámetro, no representa directamente uno de los 40 niveles de prioridad en el que se ejecutan los procesos de usuario; sino una conversión al rango [-19..0..20] de los mismos. Los valores de este nuevo rango se denominan valores

nice. De esta manera, los procesos con mayor prioridad serán aquellos que tengan un nice negativo. En este caso la terminología empleada (nice) hace referencia a que cuanto mayor es este valor para un proceso, más *amablemente* se comporta con el resto -al reducir su prioridad-. Para consultar el nice de un proceso de usuario desde el planificador, pueden emplearse las funciones `task_nice()` y `task_prio()` –definidas en kernel/sched.c–.

Prioridades Dinámicas y Prioridad Efectiva

Tal y como se mencionó al inicio de esta sección, el planificador de Linux siempre favorece a los procesos interactivos frente a los intensivos en CPU. Para implementar esta política, el planificador altera dinámicamente las prioridades los procesos; aumentando la de los primeros (disminuyendo su nivel de prioridad) y reduciendo la de los segundos (incrementando su nivel).

Este ajuste dinámico permite que los procesos se muevan entre los cinco niveles de prioridad superiores e inferiores con respecto al nivel impuesto por la prioridad estática inicial. Este valor con rango [-5...5] se denomina prioridad dinámica de un proceso. El nuevo valor de la prioridad efectiva –suma de prioridades estática y dinámica– se almacena en el campo prio de la estructura `struct task_struct` de cada proceso. Su valor se emplea como índice de acceso a los arrays de prioridad, tanto para procesos de tiempo real como de usuario.

Para saber qué procesos deben ser penalizados o beneficiados en lo que a aspectos de prioridad se refiere, el planificador emplea una heurística basada en el tiempo medio que cada proceso pasa suspendido -muy probablemente en operaciones de E/S-. De este modo, cuanto mayor sea la media de un proceso, mayor será la reducción en sus niveles de prioridad -hasta un límite de 5 niveles por debajo de su prioridad estática-. Este tiempo medio se guarda para cada proceso en el campo `sleep_avg` de su estructura `struct task_struct` asociada.

Cuando un proceso se despierta, el tiempo que ha pasado suspendido pasa a formar parte de su media. Este tiempo es contabilizado en la función `recalc_task_prio()`, que se invoca desde funciones como `activate_task()` y `schedule()`. Sin embargo, esta función no incrementa el valor de la media si el proceso suspendido permanecía en estado `TASK_UNINTERRUPTIBLE`. En este caso, dicho proceso podría ser un procesos CPU-bound que ocasionalmente se suspendió en una operación de E/S.

Finalmente, `recalc_task_prio()` invoca a `effective_prio()` para proceder a la actualización de la prioridad del procesos en el campo prio de su estruc-

tura `struct task_struct`. La función `effective_prio()` además de ser invocada desde `recalc_task_prio()`, también lo es desde `task_running_tick()`. Su funcionamiento básico es realizar una correspondencia entre el tiempo medio que un proceso pasa suspendido (`[0..MAX_SLEEP_AVG]`) y el rango de prioridades dinámicas [-5..0..+5]. La constante `MAX_SLEEP_AVG` (definida en kernel/sched.c), cuyo valor por defecto es 10ms, indica el valor máximo que puede alcanzar `sleep_avg`.

Esta modificación dinámica de la prioridad sólo afecta al 25 % de los niveles de prioridad, perteneciente a los procesos de usuario. Sin embargo, se consigue respetar la prioridad de los procesos impuesta por el usuario (nice). Por ejemplo, un proceso interactivo con nice +19 (nivel de prioridad 139) nunca podrá expropiar a un proceso intensivo en cálculo que tenga un nice 0 (nivel 120). Por otra parte, un proceso CPU- bound de nice -20 (nivel 100), nunca será expropiado por uno interactivo de nice 0.

Equidad al crear y destruir procesos

Un proceso recién creado siempre recibe como prioridad inicial, la prioridad estática de su proceso padre. Cabe destacar que la función `wake_up_new_task()`, invocada durante el proceso de creación del proceso, realiza una reducción del `sleep_avg` tanto del proceso padre como del proceso hijo recién creado. Esto impide situaciones de denegación de servicio, provocadas por procesos interactivos con valores altos de `sleep_avg`, que lleven a cabo la creación de muchos procesos hijos. Por otra parte, cuando una tarea muere, si su media es menor que la de su proceso padre, entonces la media de este ultimo se reduce de igual modo. Este procesamiento se realiza en la función `sched_exit()`.

2.4.2. Timeslices

El timeslice de un proceso es el tiempo que éste puede permanecer en ejecución en el procesador antes de ser expropiado. En general, el uso de un valor por defecto como timeslice para todos los procesos no constituye una decisión muy adecuada. El uso de timeslices de muy larga duración reduce la interactividad del sistema e incluso puede dar lugar a situaciones de inanición del resto de procesos. Por otro lado, el uso de un timeslice demasiado corto puede reducir el rendimiento de manera significativo, debido a la introducida por un excesivo número de cambios de contexto.

Cabe destacar que los procesos CPU-bound requieren timeslices largos para mantener sus datos en la cache por más tiempo, mientras que a procesos

los I/O-bound -al llegar frecuentemente a situaciones de suspensión- puede serles asignado un timeslice de corta duración. Para la política de planificación favorezca a los procesos interactivos, sin penalizar excesivamente a los intensivos en cálculo; el planificador del Linux otorga el mismo timeslice a todos los procesos inicialmente, pero trata de manera preferente a los procesos I/O-bound. A diferencia de los procesos CPU-bound, cuando un proceso I/O- bound expira su timeslice se vuelve a encolar en el array de prioridad de tareas activas. 2.4.2.1 Equidad al crear y destruir a procesos Al crear un proceso, el timeslice sin consumir del proceso padre se divide en partes iguales entre éste y el hijo en la función `sched_fork()`. De esta manera el tiempo de ejecución total que se destina a ambos permanece constante.

Del mismo modo, cuando un proceso hijo muere, su timeslice restante se reatribuye al padre, recuperando con ello parte del tiempo de ejecución que perdió al crearlo. Este procesamiento se lleva a cabo por la función `sched_exit()`.

2.4.3. Expropiación

Una de las nuevas características de Linux 2.6 es que es un sistema operativo completamente expropiativo. Este aspecto indica que el planificador puede expropiar a un proceso independientemente del modo de ejecución en el que se encuentre (usuario o núcleo). Para llevar a cabo la implementación de un modelo expropiativo, ha sido preciso realizar profundas modificaciones en la gestión los recursos del kernel, en especial en el código de acceso de aquellas estructuras de datos que requieren mecanismos de bloqueo previos a sus manipulacions para conntrolar los accesos concurrentes. Un caso claro de estructura compartida es la runqueue.

Situaciones de invocación del planificador

En ocasiones, el kernel debe invocar la función `schedule()` sin esperar a que un proceso lo haga explícitamente. En caso de no realizar estas llamadas, un proceso podría ejecutarse de manera indefinida. Para poner en conocimiento a otros subsistemas del núcleo de las situaciones en que deba invocarse el planificador, existe un flag denominado `TIF_NEED_RESCHED`,–definido en `include/asm-*/thread_info.h`–. El valor de dicho flag es consultado por el núcleo en distintos puntos. Cuando el código de un subsistema de núcleo detecta que dicho flag esta activado, invoca la función schedule() para que seleccione a una nueva tarea a ejecutar y realice en cambio de contexto entre

la tarea actual y la nueva. En cualquier caso, es el planificador (en la función schedule()) el que realiza la desactivación de dicho flag tras seleccionar una nueva tarea a ejecutar.

El flag se activa en las funciones:

- `task_running_tick()`: Cuando se detecta que un proceso ha consumido todo su timeslice.
- `try_to_wake_up()`: Cuando se despierta a un proceso con mayor prioridad que el que está actualmente en ejecución
- `wake_up_new_task()`: Cuando se crea un nuevo proceso con mayor prioridad que el que está actualmente en ejecución

Durante un equilibrado de carga, también puede ser preciso llevar a cabo una expropiación, ya que puede requerirse la migración a una CPU de una tarea con más alta prioridad que la que actualmente ejecuta en ella.

Existen una serie de funciones -definidas en include/linux/sched.h- para permitir el acceso a este flag:

- `set_tsk_need_resched(p)`: activa el flag en el proceso p.
- `clear_tsk_need_resched(p)`: desactiva el flag en el proceso p.
- `need_resched()`: consulta el valor de `TIF_NEED_RESCHED` en el proceso que está en ejecución.
- `resched_task(p)`: Esta función se utiliza como alternativa a `set_tsk_ _need_resched()` cuando el núcleo se configura con la opción `CONFIG_SMP`. La razón que motiva la existencia de esta función es tratar las situaciones en las el proceso p pueda estar asignado a otra CPU distinta a la actual, por lo que preciso invocar al planificador –función schedule()– en esa CPU.

Por motivos de eficiencia en multiprocesadores, el flag no se implementa como variable global, sino como un campo perteneciente a la estructura `thread_info` asociada a cada proceso. El coste de un acceso al descriptor de proceso en entornos multiprocesador, es menor que el implicado por el acceso a una variable global a todos los procesadores. El coste implicado es aún menor especialmente si se trata del proceso current - ya que su descriptor estará en la cache del procesador-. Sin embargo, llevar a cabo expropiación no siempre resulta seguro. Tal y como se comenta en las próximas subsecciones,

el kernel expropia a los procesos sólo en determinadas situaciones en las que se satisfacen ciertas condiciones de seguridad.

Expropiación de usuario

La expropiación de usuario ocurre cuando el flag de replanificación está activado y el kernel está a punto de retornar al modo usuario para proseguir con la ejecución del programa. Esta situación puede darse tras finalizar la ejecución de una llamada al sistema, bien tras finalizar el procesamiento de una rutina de tratamiento de interrupción.

Llevar a cabo una expropiación en este momento constituirá una operación segura, ya que no se presenta ninguna situación que impida continuar ejecutando el mismo proceso o seleccionar a uno nuevo para la ejecución. El código del núcleo que se ejecuta al volver de una llamada al sistema, o de una rutina de tratamiento de interrupción, se encargará de chequear la activación del flag de replanificación del proceso actual, llamando a la función `schedule()` en el caso de requerir ejecutar una acción de expropiación. Dicho código se implementa de distinto modo según la arquitectura y está escrito en ensamblador –puede encontrarse en en arch/*/kernel/entry.S–.

Expropiación de Núcleo

En anteriores versiones de Linux,la expropiación sólo se podía llevar a cabo cuando el proceso en ejecución estaba en modo usuario. Sin embargo, en la versión 2.6 es posible expropiar a un proceso que ejecute en cualquier modo, siempre y cuando la operación sea segura. El kernel debe estar compilado con la opción `CONFIG_PREEMPT`, para que la expropiación de núcleo esté habilitada.

Un fragmento de código es inseguro a efectos de expropiación, si incluye secciones de código en las que un proceso ha procedido al bloqueo de alguna estructura de datos o bien el proceso haya entrado en una sección crítica. Un procedimiento de expropiación en esta situación puede llevar al núcleo a una situación crítica de interbloqueo.

Para permitir distinguir los momentos seguros para expropiar a un proceso en modo kernel, de los momentos que no lo son; Linux utiliza los contadores de expropiación. Cada proceso cuenta con un contador denominado `preempt_count` que se aloja en su estructura `thread_info` asociada. El contador comienza valiendo cero y se incrementa cada vez que el proceso adquiere un cerrojo (lock), decrementandose cuando dicho cerrojo se libera. Cuando

dicho contador vale cero, el planificador puede llevar a cabo una acción de expropiación segura. Las macros proporcionadas por el núcleo para manipular los contadores de expropiación son las siguientes: `inc_preempt_count()`, `preempt_count()` y `dec_preempt_count()`. Estas macros están definidas en definidas en include/linux/preempt.h

En la nueva implementación de Linux que soporta la expropiación en modo usuario, el código del kernel consulta los valores del contador `preempt_count` y el flag `TIF_NEED_RESCHED` en las situaciones de retorno de una interrupción que implican regresos al modo núcleo:

- `TIF_NEED_RESCHED` está activado y `preempt_count` tiene valor cero: En este caso existe un proceso de mayor prioridad que el actual listo para ejecutar y debe realizarse una acción de expropiación.
- Si `preempt_count` es mayor que cero: En esta situación el proceso actual posee algún cerrojo. Llevar a cabo una operación de sincronización en este caso constituye un procedimiento inseguro.

Desactivar la expropiación

Linux permite desactivar y reactivar la expropiación de núcleo mediante las macros `preempt_disable()` y `preempt_enable()`. El funcionamiento de estas macros para impedir la expropiación, se basa en modificar el valor de `preempt_count` del proceso actual. La macro `preempt_enable()` además de reactivar la expropiación, consulta el flag `TIF_NEED_RESCHED` e invoca al planificador en caso de estar activado.

Estas macros son utilizadas para implementar la expropiación de núcleo, y se usan especialmente en las funciones de bloqueo `spin_lock()` y `spin_unlock()`, empleadas para proteger secciones críticas en el código. Para conocer más detalles sobre cómo proteger el código del núcleo de situaciones de expropiación no deseadas, puede consultarse el fichero Documentation/preempt-locking.txt presente en la documentación adjunta al código fuente del kernel.

2.4.4. Planificación en Tiempo Real

El planificador de Linux también proporciona de manera adicional una soporte para la planificación en tiempo real suave (soft real-time). El objetivo de este tipo de planificación es intentar cumplir los plazos de ejecución de los procesos (deadline) pero sin ofrecer garantía de ello. El rango de prioridades

de los procesos de tiempo real es el comprendido entre 0 y `MAX_RT_PRIO-1` -por defecto los primeros cien niveles-. La prioridad de este tipo de procesos hace que siempre expropien a los de usuario.

Esquemas de planificación

Linux emplea diferentes esquemas o políticas de planificación dependiendo del tipo de proceso. Para los procesos de usuario, se emplea `SCHED_NORMAL` –esquema empleado por defecto–. Sin embargo, para procesos de tiempo real se dispone de diferentes políticas:

- `SCHED_FIFO`: es una planificación de tipo FIFO (first-in-first-out) en la que no existen timeslices. En este esquema cada proceso se ejecuta hasta que se suspende o cede explícitamente el procesador. Cuando se produce esta situación ante la existencia de más de un proceso `SCHED_FIFO` listo para ejecutar, se elige al de mayor prioridad (menor nivel).
- `SCHED_RR`: Este esquema a diferencia del anterior sí está basado en timeslices. Cuando un proceso expira su timeslice, pasa al final de la cola de su nivel de prioridad. El siguiente proceso listo para ejecutar de la cola de más alta prioridad es el elegido para proseguir, de modo que se sigue un modelo de planificación tipo round-robin. Los procesos con este esquema de planificación tienen menos prioridad que los `SCHED_FIFO` y por tanto pueden ser expropiados por estos últimos.

El esquema de planificación de cada proceso se almacena en el campo policy de la estructura `struct task_struct`.

Cuando un proceso tiene un esquema de planificación de tiempo real, el planificador no emplea el mecanismo de prioridades dinámicas, sino que opera de manera estática con el valor inicial del campo prio. Con esto se asegura que se respetará de forma estricta su nivel de prioridad. Finalmente, cabe destacar que la llamada al sistema nice() no tiene ningún efecto sobre este tipo de procesos.

Cambiando las prioridades de los procesos de Tiempo Real

Linux proporciona una conjunto de llamadasal sistema para poder modificar y consultar la prioridad y el esquema de planificación de los procesos de tiempo real. Esto es así porque nice() sólo puede usarse con los procesos de usuario. Estas llamadas al sistema se resumen en la tabla 2.2.

Llamadas al sistema	Descripción
`sched_setscheduler()`	Establece el esquema de planificación y la prioridad de un proceso.
`sched_getscheduler()`	Devuelve el esquema de planificación de un proceso.
`sched_setparam()`	Establece la prioridad de un proceso de tiempo real.
`sched_getparam()`	Devuelve la prioridad de un proceso de tiempo real.
`sched_get_priority_max()`	Devuelve la mayor prioridad para un esquema de planificación.
`sched_get_priority_min()`	Devuelve la menor prioridad para un esquema de planificación.

Cuadro 2.2: Llamadas al Sistema para la planificación en tiempo Real

Al igual que ocurre con `nice()`, el rango de los valores de prioridad retornados o pasados como parámetros a estas llamadas al sistema no son los mismos que los usados internamente por el planificador. Estos valores son el resultado de una conversión de los valores internos al rango `[1..MAX_USER_RT_PRIO-1]`, donde el valor 1 se asocia a un proceso de tiempo real de menor prioridad y `MAX_USER_RT_PRIO-1` el de mayor. El resultado de dicha conversión se almacena en el campo `rt_priority` de la estructura `struct task_struct` de cada proceso y sólo toma el valor 0 si ese proceso se ejecuta con el esquema de planificación `SCHED_NORMAL`. `MAX_USER_RT_PRIO` es una constante definida en `include/linux/sched.h` y por defecto vale lo mismo que `MAX_RT_PRIO` (100). Sin embargo, si la primera tiene un valor menor, permite reservar los primeros niveles de prioridad a los kernel threads (procesos que sólo se ejecutan en modo kernel).

Para más detalles sobre estas llamadas al sistema, se puede consultar las páginas de manual de Linux.

Capítulo 3

Linux, MT, CMP y CMT: situación actual

3.1. Introducción

Los avances en la tecnología de los semiconductores durante los últimos años ha llevado a la inclusión de múltiples cores de ejecución en un mismo chip. Este tipo de arquitectura se conoce como multi-core (MC) o multiprocesador en chip (Chip Multi Processing CMP). Cualquier aplicación optimizada que escale correctamente en SMP obtendrá beneficios inmediatos de una arquitectura con múltiples cores. Aunque la aplicación esté formada por un único hilo de ejecución, un entorno multiprogramado puede beneficiarse de la existencia de múltiples cores de ejecución.

La familia 2.6.x del kernel Linux – que presentan una mejor escalabilidad SMP en comparación con los kernels 2.4.x– está preparada para este nuevo tipo de arquitecturas. Sin embargo, cuestiones como la compartición de recursos por parte de los cores de ejecución o reducciones en el consumo, motivan la necesidad de proponer nuevas soluciones que optimicen el rendimiento o la productividad sin degradar otros aspectos como la calidad de servicio (QoS) o el consumo.

En las plataformas basadas en procesadores multi-core, cada chip está compuesto por más de un core. Cada core tiene sus propios recursos (estado arquitectónico, registros, unidades de ejecución, uno o varios niveles de cache,...). Los recursos compartidos entre los cores de un mismo chip varían según la implementación. La mayoría de las implementaciones CMP existentes comparten el último nivel de cache y los recursos del front side bus (FSB). Los

recursos que se encuentran duplicados en las arquitecturas CMP, permiten la ejecución simultánea de varios hilos, incrementando la productividad del sistema.

Por otra parte, los procesadores multi-core también ofrecen gran densidad del sistema, así como un mejor rendimiento por vatio comparado con los chips monoprocesador a los que sustituirían. No obstante, la presencia de recursos compartidos– como el último nivel de cache, FSB, o los estados de control de consumo– entre los cores de la CPU que residen en un mismo chip, conlleva desafíos adicionales.

La mayor parte de las estrategias de programación para desarrollo sobre arquitecturas CMP emanan del entorno SMP. Las mejoras que han sido realizadas en el kernel Linux 2.6 para mejorar la estabilidad sobre arquitecturas SMP, pueden aplicarse al entorno CMP. Sin embargo, la presencia de los recursos compartidos entre cores de un mismo chip – arriba enumerados– requiere de nuevas soluciones para gozar de una óptima sintonización con la arquitectura. Conseguir llevar a cabo estos desafíos implicará una reducción en la contención por compartición de recursos así como una mejora en el rendimiento pico.

Actualmente el kernel Linux, desde su versión 2.6.19, incorpora diversas características específicas para arquitecturas CMP. La principal de ellas es la inclusión de un nuevo tipo de dominio de planificación, creado con el objetivo de diferenciar los múltiples cores de un chip de los SMPs. Con este nuevo dominio de planificación se pone en conocimiento al sistema operativo de la existencia de recursos compartidos entre los cores de ejecución.

3.2. Detección del multi-threading hardware (MT) y la topología multi-core en Linux 2.6

La existencia de múltiples unidades hardware de ejecución de hilos (MT) o cores de ejecución (CMP) en el mismo chip se presentan al sistema operativo como múltiples procesadores lógicos, de manera similar a la que aparecerían en un entorno SMP. Por ejemplo, el procesador Dual-Core Xeon 7100 series de Intel proporciona cuatro procesadores lógicos por cada chip en la plataforma, estando constituidos por dos cores cada uno con tecnología hyperthtreading.

El kernel Linux exporta la información sobre la topología MT y CMT a las aplicaciones por medio de dos mecanismos. La información sobre la topología

puede ser utilizada por las aplicaciones para numerosos propósitos como los controles de licencia, o el lanzamiento de procesos por parte de los usuarios a grupos específicos de threads, cores o chips, según sus requisitos de uso de recursos, para mejorar la productividad del sistema.

3.2.1. Exportando información de la topología a través del sistema de ficheros /proc

La entrada `/proc/cpuinfo` proporciona información sobre las diferentes CPUs en el sistema. Este fichero también exporta la información de la topología multi-core y multi-threading tal y como es percibida por el sistema operativo. Este es el método tradicional para exportar la información de la topología de la máquina, y fue creado inicialmente cuando se introdujo el multithreading simultáneo (hyperthreading). La existencia de el literal `ht` en el campo flags de `/proc/cpuinfo` indica que el procesador soporta los MSRs (Model/Machine Specific Registers) que informan sobre la existencia del HT o la capacidad multi-core de la máquina. Los campos adicionales de los registros de la CPU de /proc/cpuinfo (listados más abajo) proporcionan información más precisa sobre la topología tal y como es vista por el sistema operativo:

- `physical id` : Identificador del chip del la CPU lógica.
- `siblings`: Número total de procesadores lógicos, incluyendo threads hardware y cores, en el chip actualmente en uso por el sistema operativo.
- `cpu cores`: Número total de cores del chip que se encuentran actualmente en uso por el SO.
- core id: Identificador del core de la CPU lógica.

Las aplicaciones optimizadas para procesadores con HT, analizan los flags `ht`, `physical id` y `siblings` en `/proc/cpuinfo` para identificar los chips en el sistema. Esta identificación del chip tendrá múltiples podrá emplearse para múltiples propósitos como la implementación de sistemas de control de licencia o la asociación de un determinado thread a un chip. Estas aplicaciones no requerirán ningún cambio y seguirán funcionando tanto en sistemas multicore (CMP) como multithreading (MT), si solamente requieren conocer las CPUs lógicas asociadas a chips concretos.

Para identificar si el chip es multithreading puro (SMT), multicore-puro (CMP) o una combinación de ambos (CMT), entonces –además de la existecia del flag `ht`–las aplicaciones necesitarán analizar todos los campos arriba mencionados. Las siguientes condiciones pueden utilizarse para identificar las capacidades de la CPU mostradas al sistema operativo:

- `siblings == cpu cores >= 2`: Indica que el chip es multi-core y no tiene multithreading simultáneo
- `siblings >= 2 && cpu cores == 1`: Indica que el chip implementa multithreading simultáneo y está formado por un único core.
- `siblings > cpu cores > 1`: Indica que el chip está formado por más de un core cada uno de ellos con multithreading simultáneo.

Para que una aplicación pueda construir la topología completa para conocer la naturaleza de cada uno de los procesadores lógicos, es necesario analizar el valor de los campos `physical id` y `core_id` de la entrada /proc/cpuinfo. Para cuatro CPUs lógicas, si el valor de ambos campos es el mismo, entonces todas ellas pertenecen al mismo core (SMT siblings). Si tienen el mismo `package id` y distinto `core id`, entonces pertenecerán al mismo chip (core siblings).

3.2.2. Exportando información de la topología a través del sysfs

Las versiones más recientes del kernel Linux, como por ejemplo la 2.6.17, también permiten acceder a la información sobre la topología a través del sysfs. Este mecanismo es más simple y rápido comparado con el mecanismo relativo a la entrada del sistema de ficheros proc. Los ficheros pertenecientes a la entrada /sys/devices/system/cpu/cpuX/topology/ (listados más abajo), proporcionan la información completa sobre la topología de la máquina:

- `physical_package_id`: Identificador del chip de la CPU lógica
- `core_id`: Identificador de core de la CPU lógica
- `core_siblings`: Máscara de todas las cpus lógicas del chip
- `thread_siblings`: Máscara de todas las cpus lógicas del core

- `HW(core_siblings) == HW(thread_siblings) >= 2`: Indica [1] que el chip soporta multithreading simultáneo y está formado por un único core.
- `HW(core_siblings) >= 2 && HW(thread_siblings) == 1`: Indica que el chip está formado por más de un core pero no soporta multithreading simultáneo.
- `HW(core_siblings) > HW(thread_siblings) > 1`: Indica que el chip está formado por más de un core cada uno con multithreading simultáneo.

3.3. Planificador de Linux para CMP

El planificador de Linux, perteneciente a versiones del kernel comprendidas entre la 2.6.16 y la 2.6.18, es consciente de los dominios de planificación SMT, SMP y NUMA. En un sistema CMP, el kernel que tiene capacidades de detección multi-core ubicará en el dominio de planificador SMP a los cores y a los chips indistintamente. En el caso de requerir la inclusión un nuevo dominio de planificación para CMPs, deberían existir ciertos beneficios que lo motivaran. Estos beneficios se describen en la siguiente sección.

3.3.1. Aspectos de mejora del rendimiento pico

En una implementación de CMP donde no existieran recursos compartidos entre cores de un mismo chip, los cores serían similares a los procesadores físicos en un entorno SMP. En este caso el sistema operativo no tendría ninguna tarea más que realizar en lo que respecta al rendimiento.

Sin embargo, en la mayoría de las implementaciones multicore, con el objeto de conseguir el máximo aprovechamiento de los recursos y de hacer más eficiente la comunicación entre los cores de un mismo chip; dichos cores comparten algunos recursos. Por ejemplo, el procesador Intel Core Duo tiene está formado por dos cores que comparten el segundo nivel de cache. En este caso, el planificador del sistema operativo debería planificar tareas de tal modo que redujera la contención de recursos, maximizando el uso de los mismos y la productividad del sistema.

[1] El peso hamming (Hamming Weight,HW) es el número de bits con valor uno de la mascara de cpus del chip. Permite calcular las capacidades del mismo.

Como ejemplo (Figura 3.1), se considera un hipotético sistema formado por dos chips que comparten el FSB (front side bus). Cada chip está formado por dos cores que comparten el último nivel de cache y la cola del FSB. También se supone que en el sistema existen dos tareas totalmente independientes– no establecen ningún tipo de comunicación– y listas para ejecutar, y el planificador las sitúa en los cores 0 y 1 del primer chip dejando, por tanto, inactivo (idle) el segundo chip. En este escenario las tareas planificadas en el primer procesador físico competirán por el último nivel de cache compartido entre cores, pudiendo dar lugar a una productividad inferior a la deseada. Por otra parte los recursos del segundo chip permanecen infrautilizados. Esta decisión de planificación no es la adecuada desde el punto de vista del aprovechamiento de los recursos hardware.

La decisión óptima de planificación sería planificar las dos tareas en diferentes chips. Esto permitirá que cada tarea tenga acceso exclusivo al último nivel de cache y a una compartición justa del ancho de banda del FSB, dando lugar a un aprovechamiento más adecuado de los recursos. Por tanto, en los casos en los que los cores de un chip compartan recursos y el sistema esté en una situación de baja carga, el planificador necesitará distribuir la carga de manera razonable entre todos los chips para alcanzar el rendimiento pico.

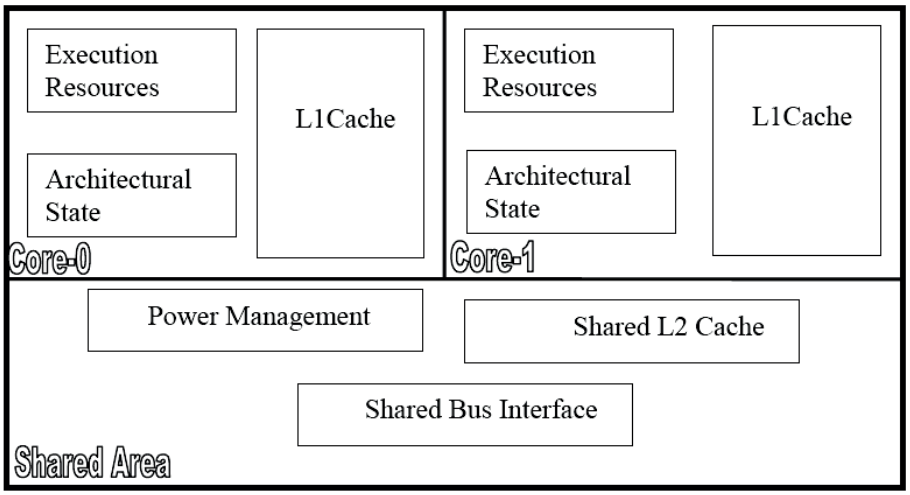

Figura 3.1: Implementación MoC con L2 e interfaz de bus (FSB) compartida

3.3.2. Aspectos de reducción del consumo

La gestión del consumo es un aspecto clave en los procesadores actuales de todos los segmentos del mercado. Existen diferentes mecanismos de ahorro de consumo como los P-estados y C-estados empleados por los procesadores de Intel. ACPI (Advanced Configuration and Power Interface) introduce el concepto de los estados de potencia (power states) de los procesadores (C0,C1,C2,C3,...Cn). El estado CO de potencia es un estado de potencia activo en el que la CPU ejecuta instrucciones. Los estados de potencia del C1 al Cn son estados del procesador en los que éste permanece dormido (idle), y en los que consume menos potencia, disipando menos calor.

Mientras el procesador permanece en estado C0, ACPI permite modificar el rendimiento del procesador a través de los estados de rendimiento (performance states). Mientras la mayor parte de la gente piensa en los P-estados en términos de reducción de la frecuencia del procesador cuando éste no se encuentra muy ocupado, en realidad consisten en una combinación más compleja en la que se producen reducciones de la frecuencia y el voltaje. Mediante los P-estados, puede conseguirse que la CPU consuma menos potencia proporcionando menos rendimiento en estado C0 (en ejecución). Dado un P-estado, la CPU puede realizar una transición a estados C superiores cuando se encuentra inactiva. En general cuanto mayor son los pares de P-estado y C-estado del procesador /core menor potencia consume y menor calor disipa.

3.4. Implicaciones de CMP en estados P y C

En una configuración CMP, habitualmente todos los cores de un mismo chip comparten el mismo voltaje, ya que existe un único regulador de voltaje por socket de la placa madre. De este modo las transiciones entre P-estados entre los cores– que tendrán impacto tanto en la frecuencia como en el voltaje del procesador– han de ocurrir al mismo tiempo. Este mecanismo de coordinación entre los P-estados entre cores pueden implementarse vía hardware o software. Asimismo las solicitudes de transición entre P-estados de cores de un mismo chip también deben ocurrir al mismo tiempo, efectuando la transición al estado P destino del chip cuando dicha transición garantice no originar un P-estado incorrecto o no óptimo. Si un core está consumiendo un 100 % de CPU ejecutando una tarea, esta coordinación asegurará que otros cores que se encuentren inactivos en el mismo chip no podrán entrar en P-estados inferiores, dando lugar a que el chip permanezca en el P-estado más alto para obtener un rendimiento óptimo. En general, esta coordinación

asegurará que la frecuencia del procesador será la relativa al P-estado inferior numéricamente hablando, (frecuencia y voltajes mas elevados) y dicha frecuencia será la misma para todos los procesadores lógicos del chip.

3.4.1. Sincronización de C-estados

En una configuración CMP habitual, el chip del procesador puede estar partido en diferentes bloques. Un bloque para cada core de ejecución y un bloque común representando los recursos compartidos entre todos los cores. Como los cores operan de forma independiente, los bloques de cada core pueden transitar de forma independiente entre los C-estados. Por ejemplo un core puede estar en C1 o C2 mientras otro ejecuta el código de una aplicación en estado C0. El bloque común estará en el mínimo estado en el que se encuentren los bloques pertenecientes a cada core, el estado C de máxima potencia. Por ejemplo, si un core esta en C3 y otro está en C0 entonces el bloque compartido estará en estado C0.

3.5. Política de planificación para ahorro de consumo

Se toma como ejemplo el mismo hipotético sistema anteriormente considerado, el cual estaba compuesto de dos procesadores físicos con dos cores cada uno que comparten el último nivel de cache y los recursos del FSB. En el escenario, en el que existen únicamente dos tareas en el sistema listas para ejecutar –de forma similar al ejemplo previo– el rendimiento pico podrá lograrse cuando estas dos tareas se planifican para ejecutar en cores de distintos chips. Sin embargo para lograr dicho rendimiento pico, y debido a la necesidad de la coordinación entre los P-estados, los cores donde ejecuten dichas tareas deben residir en el P-estado de máxima potencia (P0). Como todos los procesadores lógicos (o cores) del chip comparten el mismo P-estado, existirían dos cores inactivos con P-estado 0, consumiendo una potencia innecesaria. De manera similar el bloque compartido entre ambos chips residirá en el C-estado 0 (de máxima potencia) , por la existencia de un core que ejecuta instrucciones (en C0). Esta situación provoca que desde el punto de vista del ahorro de consumo, la planificación realizada no sea óptima.

Alternativamente, si el planificador elige el mismo chip para lanzar ambas tareas –una en cada core–, el otro chip que se encontrará completamente

inactivo, realizará una suave transición a los estados P y C superiores, que involucran un menor consumo de potencia. Sin embargo, como los cores del mismo chip comparten el último nivel de cache, el hecho de planificar las dos tareas a dichos cores no conducirá a un comportamiento óptimo desde el punto de vista del rendimiento.

De este modo, el impacto en el rendimiento dependerá del comportamiento del las tarear de los recursos hardware entre los cores. En este ejemplo en particular, si las tareas no son intensivas en el uso de memoria/cache, el impacto en el rendimiento será mínimo. En general es posible ahorrar más potencia con un mínimo impacto en el rendimiento, siguiendo la estrategia de planificación que ubica las dos tareas en los cores de un mismo chip [15].

Para implementar en Linux la estrategia de ahorro de consumo anteriormente descrita, el planificador tiene que poder diferenciar cores y chips. Para ello se añade un dominio de planificación, representando los cores de una arquitectura CMP, que permitirá hacer esta distinción para la toma de decisiones de equilibrado de carga.

El dominio de planificación MC (multi-core) que indica la existencia del último nivel de cache compartido entre los cores pertenecientes al mismo chip, ha sido incoporado al kernel 2.6.17. La figura 3.2 muestra la política de obtención del rendimiento pico que se encuentra activada por defecto en el kernel 2.6.17. En esta figura se muestra a cuatro tareas ejecutando en un sistema con dos chips, cada uno con dos cores con LLC (last level cache) compartido, y cada core con dos threads lógicos (multithreading simultáneo). Para el sistema operativo, la máquina consta de ocho procesadores lógicos. El equilibrador de carga actúa en el dominio MC (multi-core) para el primer chip, dando lugar a una distribución equilibrada de la carga entre los cores.

La política de ahorro de consumo fue introducida en la versión 2.6.18-rc1 de Linux. Las entradas del sysfs `sched_mc_power_savings` y `sched\_smt_ _power_savings` en /sys/devices/system/cpu/ controlan la política de ahorro de consumo para arquitecturas multi-core/multithreading. Cuando la política de ahorro de consumo está habilitada bajo condiciones de baja carga, el planificador minimizará los chips/cores que asuman la carga del sistema. Esta acción permitirá un ahorro considerable de consumo, y el impacto en el rendimiento dependerá de las características de los procesos en ejecución.

La figura 3.3 muestra el equilibrado de carga para la política mejorada de ahorro de consumo, con cuatro tareas en ejecución. El sistema consta de dos chips cada uno con cuatro cores. El equilibrado de carga entre los dos chips (dominio de planificación SMP), provoca el movimiento de todas la tareas a los cores del mismo chip, permitiendo un ahorro en el consumo y con un

impacto asociado en el rendimiento que dependerá del comportamiento de las tareas y de los recursos compartidos entre los cores.

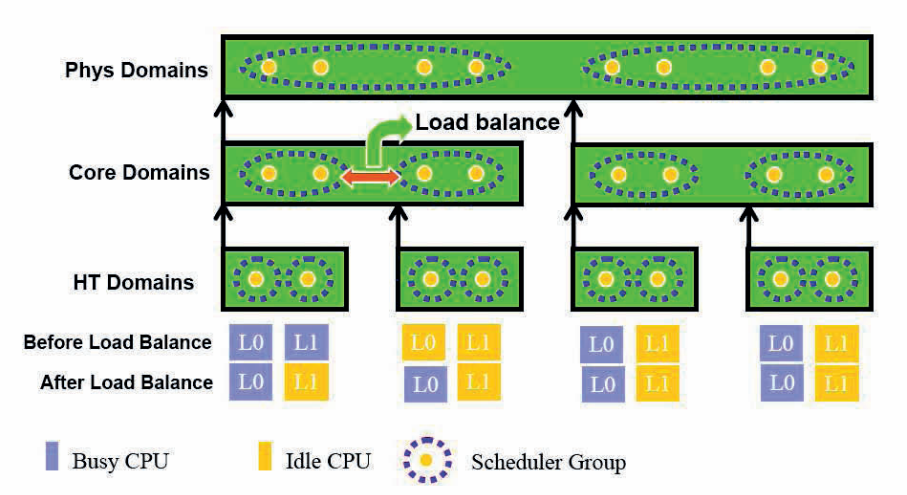

Figura 3.2: Demostración de la política de obtención del rendimiento pico en un sistema CMT (Chip Multithreading) con dos chips dual-core, y con cada core con mulithreading simultáneo de dos vías (8 procesadores lógicos).

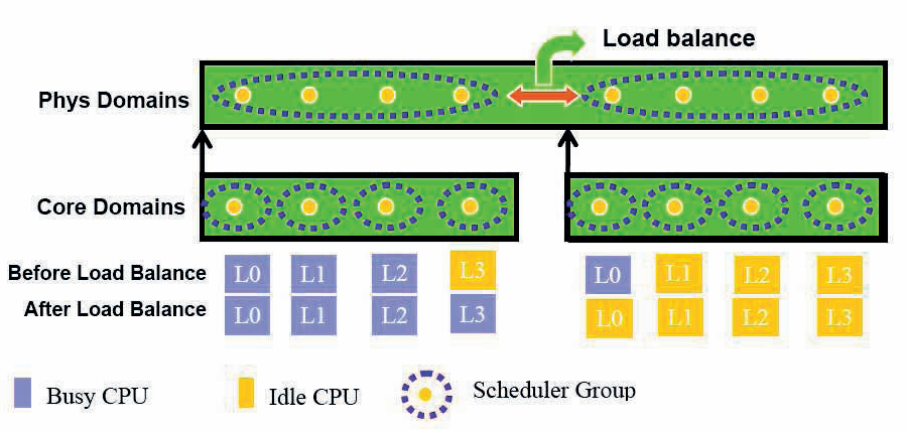

Figura 3.3: Demostración de la política de ahorro de energía en una arquitectura SMP/CMP formada por dos chips, cada uno formado por cuatro cores.

Capítulo 4

Calidad de servicio en arquitecturas CMP

4.1. Problemática

La mayor parte de las implementaciones de procesadores multicore presentan uno o más niveles de cache compartidos entre los cores. Para comprender la ventaja de este diseño es preciso tener en cuenta el modelo de aplicaciones que se ejecutan actualmente en servidores. Este tipo de aplicaciones -como web services, servidores de aplicaciones, servidores web o de bases de datos- están constituidas frecuentemente por múltiples hilos de control que ejecutan breves secuencias de operaciones enteras, y saltos condicionales. Por otra parte, en los entornos de escritorio suele ser común la ejecución simultánea de múltiples tareas en segundo plano -como servicios del sistema, cliente de correo, antivirus- con otras tareas de naturaleza interactiva -como un navegador web o un procesador de texto-.

En procesadores de la microarquitectura Core [16], como el procesador Intel Core 2 Duo –formado por dos cores– existe un segundo nivel de cache (L2) compartido por cada cores. En el caso de haber optado por un diseño en el que cada core tuviera un segundo nivel de cache independiente (o dividido de forma exclusiva), dicho diseño podría dar lugar a situaciones no deseadas en casos de compartición de datos por parte de ambos cores.

Con L2 compartido se optimiza el uso de los recursos de la cache. Por otra parte, el hecho de disponer de un segundo nivel de cache compartido, este diseño permite que cada uno de los cores pueda utilizar dinámicamente hasta un 100 % de la cache de L2 disponible. Cuando un thread que ejecute en un

core tenga requisitos mínimos de cache, los threads de otros cores pueden incrementar su porcentaje de uso de L2, reduciendo los fallos de cache en relación a una implementación multicore con cache de segundo nivel dividida o única por core.

Sin embargo, el segundo nivel de cache compartido puede dar lugar a conflictos por dicha compartición en situaciones de uso intensivo de L2, provocando que el rendimiento efectivo del sistema se distancie notablemente del rendimiento ideal.

Por otra parte, la mayoría de los procesadores –al igual que el procesador Intel Core 2 Duo– carecen de mecanismos a nivel hardware para fijar la fracción de cada recurso compartido que se destina al uso por parte de cada procesador lógico[1] (core). Esto provoca que hilos de distinta prioridad que ejecuten en distintos procesadores lógicos con recursos compartidos no gocen de un reparto de los recursos acorde a su prioridad. Por esta razón, resulta necesario que los sistemas operativos que ejecuten en arquitectura CMP deban ser conscientes de la compartición de recursos para tenerla en cuenta en el diseño de sus políticas de calidad de servicio (QoS), así como lograr que la productividad del sistema no se vea degradada para conseguir tal fin.

4.2. Soporte para la calidad de servicio en Linux 2.6

El planificador de Linux (en su versión 2.6.x) gestiona colas de ejecución independientes por procesador llevando a cabo un esquema de planificación distribuida . Por otra parte, emplea un mecanismo de equilibrado de carga de las colas de ejecución para ejercer un control global sobre la planificación. En entornos multiprocesador, este diseño presenta mejor escalabilidad que el diseño empleado en la versión previa de Linux (versión 2.4), en el que existe una única cola de ejecución compartida por todos los procesadores.

En una misma cola de ejecución coexisten procesos de distinta prioridad. El planificador reparte el tiempo de CPU en función de la prioridad de cada proceso, de tal manera que un proceso con más alta prioridad goza de un trato preferente con respecto a los procesos de inferior prioridad. De este modo, Linux ofrece soporte de calidad de servicio para el conjunto de procesos que residen en una misma cola de ejecución.

[1] Cabe destacar que el IBM Power5 [8] permite asignar diferentes fracciones de recursos compartidos a cada thread. El reparto de recursos se realiza implicitamente asignando distintos ciclos de decodificación a cada thread.

No obstante, en aquellas arquitecturas multiprocesador en las que los procesadores lógicos comparten recursos internos del procesador, niveles de cache o recursos del bus del sistema con otros procesadores lógicos; el planificador de Linux no implementa ninguna política de calidad de servicio global. En este tipo de arquitecturas, Linux permite la ejecución simultánea de procesos de distinta prioridad en distintas CPUs lógicas ignorando la posible compartición de recursos existentes entre las mismas. Una excepción a esta situación son las arquitecturas SMT. El parche desarrollado recientemente para procesadores con SMT (como el Intel Pentium IV con tecnología Hyperthreading), altera el comportamiento del planificador no permitiendo ejecutar dos procesos de distinta prioridad en distintas CPUs de forma simultánea.

Adicionalmente, el diseño del planificador de Linux impone serias dificultades para llevar a cabo la implementación de mecanismos globales de calidad de servicio. Estos mecanismos han de tener en cuenta las prioridades de los procesos que ejecutan simultáneamente en las diferentes CPUs. Para acceder a esta información de modo seguro desde un procesador, es necesario realizar bloqueos de las colas de ejecución del resto de procesadores y así obtener los descriptores de los procesos que permanecen actualmente en ejecución para conocer dichas prioridades. Estas acciones implican gran sobrecarga en el sistema limitando la escalabilidad del modelo de planificación introducido recientemente en Linux.

4.2.1. Modelos de calidad de servicio en Linux/CMP

Existen varias aproximaciones para implementar mecanismos globales de calidad de servicio en la versión 2.6 de Linux:

- Aproximación exclusiva
- Aproximación optimista o basada en monitorización
- Aproximaciones basadas en caracterización

Las políticas globales de calidad de servicio que siguen una aproximación exclusiva, son aquellas que no permiten la ejecución simultánea de dos tareas de distinta prioridad en diferentes CPUs. En implementaciones de esta política sobre arquitecturas CMP, los procesos de más alta prioridad gozan de todos los recursos compartidos a su disposición. Un ejemplo de implementación de este tipo de política es el parche de SMT de Linux. El principal inconveniente que presenta este modelo en arquitecturas CMP, es la drástica reducción de la productividad del sistema.

Las aproximaciones optimistas o basadas en monitorización, permiten -por defecto- la ejecución simultánea de más de un proceso de distinta prioridad en distintas CPUs. Sin embargo, durante dicha ejecución, el sistema operativo realiza acciones de monitorización para detectar las situaciones perjudiciales derivadas de la libre utilización de los recursos por parte de cada tarea en ejecución. En el caso de que procesos menos prioritarios provoquen una disminución considerable del rendimiento de otros más prioritarios, debido al uso intensivo de uno o más recursos compartidos entre los procesadores lógicos -como la cache de segundo nivel-; el planificador deberá posponer la ejecución de dichos procesos conflictivos. Un ejemplo de implementación de este tipo de política es el planificador simbiótico para procesadores de doble núcleo (dual-core) que se ha desarrollado en este trabajo.

Finalmente, las aproximaciones basadas en caracterización realizan clasificaciones de los procesos según el porcentaje de uso de recursos compartidos. Esta información puede extraerse -a grandes rasgos- mediante un proceso de monitorización, de la tarea en cuestión, durante una ejecución en la que dicha tarea goce de todos los recursos a su disposición- por ejemplo, ejecutando sola, sin procesos que puedan interferir la medición- . A la hora de lanzar un proceso en una determinada CPU, el planificador no solo tendrá en cuenta la prioridad de dicho proceso, sino también las posibles consecuencias de llevar a cabo una ejecución simultánea con los procesos de distinta prioridad que ejecutan actualmente en el sistema. Estas consecuencias pueden deducirse gracias a la información de uso recursos almacenada para cada proceso. La propuestas de planificadores simbióticos de [12] y [17] siguen un modelo basado en caracterización. Este tipo de aproximaciones no sólo se emplean en el ámbito de la calidad de servicio sino también en el diseño de políticas cuyo objetivo sea mejora de otros aspectos como la productividad o el rendimiento del sistema.

4.2.2. El planificador SMT de Linux

En la actualidad, Linux incorpora un parche para el planificador, cuyo objetivo es dar soporte a la calidad en servicio en arquitecturas SMT. El planificador para SMT impide que dos tareas con valores *nice* diferentes,se ejecuten simultáneamente en dos procesadores lógicos de una misma CPU SMT física. Mediante el empleo de esta técnica se han podido obtener resultados experimentales que muestran una mejora del 5 % en el tiempo de ejecución del proceso más prioritario. El principal problema, que presenta esta solución, es que el planificador deja en manos del programador y del administrador del sistema la asignación a cada proceso de los valores de nice adecuados para

optimizar el rendimiento de todo el sistema.

Las principales modificaciones del código del núcleo realizadas por este parche son las siguientes:

- Define la función `cpu_and_siblings_are_idle()`, que recorre todo el conjunto de todas las CPUs hermanas- dos CPUs lógicas dentro de la misma CPU SMT- física-, para comprobar si todas se encuentran ociosas. Esta función se llama desde las siguientes funciones:
  - `can_migrate_task()`: Se invoca para chequear si una tarea puede ser migrada a una CPU física por estar ociosa. En una arquitectura SMT, para que una CPU física se considere ociosa, cada uno de los procesadores lógicos en su interior han de estarlo.
  - void `active_load_balance()` Esta función hace uso de `cpu_and_siblings_are_idle()` para saber si una CPU física está ociosa. El objetivo que persigue es evitar el desequilibrio entre CPUs físicas. La política de equilibrado de carga en SMT da prioridad a la ocupación de las CPUs no SMT con respecto a la ocupación de las cpus lógicas.
- Introduce la función `wake_priority_sleeper()` cuyo objetivo es replanificar el proceso idle. En el caso de que se hubiera producido el bloqueo de tareas, que se enviaron a dormir por razones de prioridad, esta función procederá a despertarlas. `wake_priority_sleeper()` es invocada por `scheduler_tick()` cada tick de planificador.
- Define la función `wake_sleeping_dependent()`, encargada de despertar todas las tareas dormidas por razones de prioridad en las CPUs lógicas que componen una CPU física. Esta función se invoca desde `schedule()` -función principal del planificador- cuando se detecta que una cola de ejecución se queda vacía.
- Añade la función `dependent_sleeper()`, invocada desde la función principal de planificación `schedule()`. Su funcionamento es el siguiente:
  - Comprueba si existe alguna tarea de la cola de ejecución pasada como parámetro, que pueda ejecutarse por motivos de prioridad.
  - En el caso de que esto no sea posible, se procede a la selección de tarea idle para abandonar la CPU, dejándola libre para alguna de las tareas más prioritarias en las CPUs hermanas.

  - En caso contrario, selecciona la tarea más prioritaria de la cola de ejecución y compara su timeslice con el de las tareas que estén siendo ejecutadas en las CPUs hermanas de manera simultánea. Si este timeslice es inferior 75 % al de la tarea hermana (criterio de prioridad), ninguna tarea perteneciente a esa cola de ejecución podrá entrar a ejecutar[2].
  - Finalmente, la función se encarga de replanificar aquellas tareas hermanas en ejecución cuya prioridad sea menor (es decir, con un timeslice restante inferior al 75 %) que la de la tarea seleccionada. De ese modo, se impide que la ejecución de una tarea de baja prioridad pueda afectar negativamente al rendimiento de las tareas más prioritarias.

- Define la función `cpu_to_phys_group()` que devuelve una CPU representante de la CPU física a la que está asociada cpu. La CPU que devuelve como CPU física es la primera de todas las hermanas. En el caso de que el sistema estuviera constituido por un solo procesador con SMT, tanto a la CPU 0 como a la 1 les corresponderían la CPU física 0.

Las principales acciones del planificador SMT se realizan cada vez que se va a seleccionar una nueva tarea para ejecución. En este caso, el planificador actúa del siguiente modo:

- Si existe una tarea más prioritaria - con un 75 % de timeslice superior al de la nueva tarea-, el planificador SMT impide su ejecución. En dicha situación se da paso a la ejecución del proceso idle, cuya activación hace entrar al procesador en un modo de bajo consumo. En dicho modo, todos los recursos del procesador pasan a estar completamente disponibles para el proceso de mayor prioridad que se encuentra actualmente en ejecución.

- Si la tarea seleccionada es más prioritaria que las que están en ejecución en las CPUs hermanas, las tareas de dichas CPUs con un timeslice inferior al 75 % del timeslice de la tarea seleccionada (inferior prioridad) se replanifican.

[2]El parche de SMT no utiliza directamente el valor *nice* como criterio de prioridad, ya que utiliza la duración del *timeslice* de un proceso. No obstante, existe una relación indirecta ya que el *timeslice* se asigna en función del *nice* (además de en función de otros parámetros)

Capítulo 5

Planificador simbiótico para calidad de servicio en arquitecturas CMP

Como se comentó en el capítulo anterior, la política de calidad de servicio de Linux sobre arquitecturas CMP no tiene en cuenta la existencia de recursos compartidos entre cores. Esta situación da lugar a que la ejecución de tareas de baja prioridad que hagan uso intensivo de alguno de dichos recursos compartidos, impacte negativamente en el rendimiento de tareas de más alta prioridad que ejecuten simultáneamente en distintos cores del mismo chip.

En la primera parte de este capítulo se expone la política de planificación propuesta en este trabajo, cuyo objetivo es dar soporte de calidad de servicio en arquitecturas CMP de dos vías (dos cores). Dicha implementación ha sido realizada sobre la versión de Linux kernel 2.6.21 ejecutando en un procesador Intel Core 2 Duo E6300 (1.68 GHz) con una cache de segundo nivel (L2) compartida de 2MBytes [16].

Los componentes del planificador simbiótico y los detalles de implementación e interacción con el usuario se tratan en la segunda parte del capítulo.

5.1. Funcionamiento del planificador simbiótico

El término simbiosis se utiliza actualmente para referirse a la efectividad con la que se obtiene mayor rendimiento o productividad al ejecutar múltiples hilos de manera simultánea en arquitecturas multithreading (MT o SMT). Este concepto fue utilizado inicialmente por Snavely, et al en [12] y ha estado

unido a las estrategias de planificación en arquitecturas SMT (planificación simbiótica). Sin embargo, en la era de las plataformas CMP, este concepto puede extenderse a la planificación sobre este tipo de arquitecturas; ya que sigue existiendo un notable índice de compartición de recursos (L2 cache o Front Side Bus) cuyo impacto sobre el rendimiento de las aplicaciones actuales sigue siendo crítico [13].

El objetivo de este planificador simbiótico es optimizar el rendimiento de los procesos más prioritarios que ejecutan en entornos CMP, sin que ello implique una notable degradación de la productividad global.

Las modificaciones realizadas en el planificador de Linux, para lograr los objetivos de calidad de servicio, siguen una aproximación basada en monitorización (sección 4.2.1). De este modo, a diferencia de las aproximaciones exclusivas, el planificador permite, por defecto, la ejecución de procesos de distinta prioridad simultáneamente en ambos cores. En situaciones en las que el rendimiento del proceso más prioritario se vea perjudicado por un uso intensivo de la cache de L2 por parte de un proceso menos prioritario; el planificador impide la ejecución simultánea de ambas tareas, cediendo todos los recursos compartidos (FSB y cache de L2) a la tarea más prioritaria.

Los procedimientos de monitorización se llevan a cabo haciendo uso de los contadores de monitorización del rendimiento del procesador Intel Core 2 Duo. El sistema contabiliza dinámicamente la tasa local de fallos de segundo nivel de cache y el número de instrucciones por ciclo, haciendo uso de la unidad de monitorización del rendimiento (PMU) del procesador, y almacena dichos valores en el descriptor de proceso (`struct task_struct`). Si durante una ejecución de dos tareas de distinta prioridad, se detecta una notable subida de la tasa de fallos local de L2 del proceso más prioritario –umbral configurable–; el planificador procede a la desactivación del core donde ejecuta el proceso menos prioritario.

La desactivación de un core implica la migración de todas las tareas residentes en su cola de ejecución asociada -incluida la tarea menos prioritaria causante de la desactivación- a la cola del core. Tras la desactivación de uno de los cores, Linux constará de una única cola de ejecución no vacía que almacena todos los procesos del sistema. En este caso, el planificador repartirá el tiempo de la CPU lógica activa (core activado) entre todos los procesos del sistema, teniendo en cuenta aspectos de interactividad y prioridad. Cada proceso que entre a ejecutar en estados de desactivación, podrá usar el 100 % de los recursos compartidos por ambos cores, incluido el segundo nivel de cache. De este modo, se asegura la calidad de servicio, ya que el tiempo de uso de los recursos compartidos se asigna en función de la prioridad de los

procesos.

No obstante, el mantenimiento de un core desactivado durante un largo tiempo sólo favorece a los procesos más prioritarios, llegando a degradar la productividad del sistema. Por consiguiente, las situaciones de desactivación sólo deben mantenerse durante situaciones de conflicto por compartición, durante las cuales pueden lograrse mejoras significativas del rendimiento del proceso prioritario con respecto al comportamiento natural de Linux. En situaciones libres de conflicto, el estado de desactivación sólo implica penalizaciones en la productividad, ya que el planificador mantiene parados a procesos de baja prioridad cuya ejecución no provoca una degradación del rendimiento de procesos más prioritarios.

Cuando el proceso más prioritario recupera la tasa de fallos de L2 que tenía hasta el momento (valor promedio) y, a partir de ese momento, abandona la CPU o agota su *timeslice* ; dicho core vuelve a rehabilitarse.

5.1.1. Desactivación de un core

Como se comentó en la sección previa, el planificador simbiótico realiza acciones de activación y desactivación de uno de los dos cores para resolver las situaciones de conflicto entre procesos de distinta prioridad. El proceso de desactivación de un core se implementa vía software, llevándose a cabo por sistema operativo.

Para lograr la desactivación completa de un core, es necesario migrar todas las tareas de la cola de ejecución de dicho core, a la cola de ejecución del core que permanezca activado. Antes de desarrollar un procedimiento de migración es necesario tener en cuenta los siguientes aspectos:

- Existen procesos de sistema, asociados a una CPU determinada, cuya migración a otra CPU puede tener implicaciones catastróficas, llevando al sistema a situaciones de bloqueo irrecuperables. Este tipo de procesos no migrables, son el init (con pid 1), el swapper (con pid 0) y los kernel threads (en especial, los hilos de migración).

- Linux permite fijar las CPUs en las que una tarea puede o no ejecutarse. El conjunto de CPUs en las que una tarea puede ejecutarse se denomina conjunto de afinidad. La representación interna de estos conjuntos de CPUs se lleva a cabo mediante máscaras de afinidad:

Definición 1 *La máscara de afinidad de un proceso es un número m de n bits* $\boxed{m = b_{n-1}b_{n-2}\ldots b_2b_1b_0}$ *en la cual cada bit* $b_i = 1$ *indica que la tarea puede ejecutar en la CPU i*

- Las tareas cuya migración implica trabajo extra o resulta perjudicial en este tipo de entornos, son aquellas que:
  1. Están en ejecución.
  2. No son afines a la CPU de destino
  3. Son *cache-hot* en su CPU, es decir tareas que han abandonado muy recientemente dicha CPU y por lo tanto es muy probable que se mantengan en cache datos e instrucciones de dicho proceso.

Por las razones previamente expuestas, las tareas de sistema no pueden ser migradas a la CPU opuesta. El hecho de que estas tareas permanezcan en su cola de ejecución no implica la existencia de situaciones de conflicto por compartición de recursos, ya que estas tareas permanecen la mayor parte del tiempo dormidas y su prioridad es más alta que la de procesos que no son de sistema.

Por otra parte, para lograr la desactivación efectiva del core, el proceso de migración de tareas debe ignorar cuestiones de afinidad con respecto a la CPU destino de la migración. Por lo tanto, el planificador debe permitir -incondicionalmente- la ejecución de la tarea en el core que permanezca activado, durante las situaciones de desactivación.

En el caso simétrico, durante las situaciones de reactivación, aquellas tareas sin afinidad a la CPU activada (ignorada durante la migración) deben migrarse antes que ninguna otra tarea. El resto de migraciones *necesarias* para completar la reactivación se dejan al proceso de equilibrado de carga de Linux, ya que, de este modo, las tareas migradas para garantizar el equilibrio de carga serán aquellas que hayan estado menos recientemente en la CPU.

Finalmente, cabe destacar que en las situaciones en las que el sistema operativo detecta conflictos entre procesos de distinta prioridad, ambos procesos están en posesión de su respectiva CPU. Por esta razón, antes de proceder a la desactivación, es preciso llevar a cabo una expropiación del proceso que está ejecutando en el core a desactivar. El mecanismo de expropiación empleado se basa en el uso de kernel threads de alta prioridad. La función de estos kernel threads no se limita al procedimiento de expropiación, sino que además llevan a cabo la construcción de listas de peticiones de migración

para los hilos de migración de su CPU. De este modo, el planificador simbiótico implementa el procedimiento de desactivación de los cores mediante kernel threads de desactivación, uno por cada CPU.

Selección del core a desactivar

El objetivo de la migración es detener las situaciones de conflicto entre procesos de distinta prioridad. En este caso, el proceso más prioritario debe ser el proceso beneficiado de las situaciones de desactivación. La desactivación de uno de los cores implica, en cualquier caso, que el proceso más prioritario dispone de todos los recursos compartidos a su disposición para ejecutar; pero en arquitecturas CMP la elección del core a desactivar afecta notablemente en el rendimiento de los procesos.

En el momento de la detección de una situación de conflicto, los dos procesos de distinta prioridad serán *cache-hot* [1] en su CPU.Por otra parte, en una arquitectura CMP compuesta por múltiples cores con una memoria cache (L1) independiente y un segundo nivel de cache compartido, solo existe afinidad a la cache de nivel 1. De este modo, la migración a otra CPU de un proceso cache-hot provocará que pierda su afinidad a L1, y, por lo tanto, dicha migración degradará el rendimiento del proceso. En consecuencia, como la desactivación no debe perjudicar en ningún caso al proceso más prioritario, el core seleccionado para desactivar será aquel donde ejecute el proceso menos prioritario.

Cabe destacar, que en el caso de extender esta política de calidad de servicio a arquitecturas SMT de dos vías, la elección del procesador lógico a desactivar no afectaría al rendimiento del proceso más prioritario, ya que en este tipo de arquitecturas no existe afinidad a la cache.

Detección de conflictos de compartición

Para proceder a la detección de las situaciones de conflicto por compartición de la cache de segundo nivel (L2), durante la ejecución de dos procesos de distinta prioridad; es preciso que el planificador simbiótico disponga de información sobre el comportamiento de los procesos con respecto a la cache. Gracias a los contadores de monitorización del rendimiento, el sistema puede tener acceso a cierta información que permite deducir dicho comportamiento.

[1] Aquí el término *cache-hot* hace referencia a las tareas que han abandonado muy recientemente su CPU y por lo tanto es muy probable que se mantengan en cache datos e instrucciones de dicha tarea.

Los contadores de monitorización del rendimiento del procesador Intel Core 2 Duo (microarquitectura Core), permite capturar –además de otros eventos– la cuenta de el número de instrucciones retiradas, el número de ciclos de procesador, el número de accesos al segundo nivel de cache (LLC, last level cache) y el número de fallos de acceso a L2. La herramienta de monitorización del rendimiento incluida en el planificador simbiótico permite no sólo capturar estas medidas, sino también calcular parámetros de rendimiento de alto nivel (*eventos de alto nivel*). Este mecanismo de más alto nivel -que los eventos hardware del procesador- permite representar, entre otra información, la tasa de fallos de cache, la tasa de aciertos en la predicción de saltos o el número de instrucciones por ciclo.

Para cada proceso, el planificador almacena un valor medio de la tasa de fallos local de cache de L2, calculado a partir de la información proporcionada por la herramienta de monitorización. Cuando la tasa de fallos de L2 de cache registrada en la última medición supera a este valor en tantas unidades como indique el umbral de calidad de servicio (parámetro configurable), el planificador procede a realizar una desactivación del core, por considerar esta situación como conflictiva.

El cálculo de este valor medio se realiza en función del valor medio anterior y los k últimos valores registrados de la tasa de fallos local de L2. Las razones que motivan la utilización de los últimos k valores para el cálculo de la media son las siguientes:

- Por cuestiones de espacio de almacenamiento en memoria, es inviable almacenar todas las muestras obtenidas para cada proceso.
- Por la propiedad de localidad temporal, los últimos valores registrados se aproximan más que los más a alejados en el tiempo, con respecto a los valores de la tasa de fallos local de L2 que se registrarán próximamente. Como prueba de ello, en la sección 6.1.1, pueden observarse las características obtenidas experimentalmente, que muestran la evolución en el tiempo de la tasa de fallos de cache asociada a algunos benchmarks de SPEC CPU2006.

De este modo, la fórmula asociada al cálculo de valor medio de la tasa de fallos local de L2 de un proceso en un instante de muestreo t+1 es la siguiente:

$$\text{valor\_promedio}(t+1) = \frac{\sum_{i=0}^{k-1}(\text{l2\_miss\_rate}((t+1)-i) + \text{valor\_promedio}(t)}{k+1}$$

Para almacenar en el descriptor de proceso, los últimos k valores de la tasa de fallos local de L2 asociada a un proceso se utiliza un buffer circular de longitud k. Sin embargo, para permitir que este parámetro k sea configurable en tiempo de ejecución (`stats_buffer_size`), el planificador emplea un buffer circular especial (`mc_cbuffer_t`) cuyo tamaño máximo puede modificarse durante la ejecución.

Mediante el empleo de esta métrica de detección, el planificador realiza una predicción del próximo valor de la tasa de fallos local de L2 que presentará el proceso. Una amplia diferencia del nuevo valor con respecto a la predicción, puede deberse a una situación de conflicto por compartición, o bien a la naturaleza del programa. En cualquier caso, si la diferencia entre el nuevo valor y la predicción del valor promedio, supera un umbral configurable, el planificador lleva a cabo la desactivación del core donde ejecuta el proceso menos prioritario.

5.1.2. Situaciones de reactivación

Como se comentó en la sección previa, las subidas drásticas (con respecto a la predicción) de la tasa de fallos local de L2 de un proceso, puede deberse a una situación de conflicto por compartición, o bien a la naturaleza del programa. La causa de este incremento puede conocerse al desactivar un core.

Una vez realizado el proceso de desactivación, el core permanecerá inactivo durante un periodo denominado intervalo de desactivación. Este intervalo es configurable, y está determinado por dos parámetros `min_ticks_out` y `max_ticks_out`. El parámetro `min_ticks_out` determina el número de ticks de sistema que el core permanecerá desactivado como mínimo. Por otra parte, el parámetro `max_ticks_out` establece el número máximo de ticks de sistema que el core permanecerá desactivado.

Adicionalmente, se define el intervalo de calidad de servicio del planificador simbiótico como sigue:

Definición 2 *Sea c el valor promedio (o predicción) obtenido anteriormente a la desactivación y ε el umbral de calidad de servicio introducido por el usuario, se define el intervalo de calidad de servicio como* $[0, c + \varepsilon]$

En el instante en el que se inicia el proceso de desactivación, el cálculo y actualización del valor promedio se deshabilita para la tasa local de fallos de L2 del proceso más prioritario. El proceso de actualización se reactiva una

vez transcurrido un número de ticks mayor o igual a `min_ticks_out` desde la finalización del proceso de desactivación del core.

Cuando han transcurrido, al menos `min_ticks_out` desde la desactivación efectiva del core, el planificador actúa del siguiente modo:

1. Si ha transcurrido un número de ticks superior o igual a `max_ticks_out` el core desactivado vuelve a rehabilitarse

2. Si el último valor -proporcionado por la herramienta de monitorización interna- de la tasa de fallos local de L2 del proceso prioritario causante de la desactivación, pertenece al intervalo de calidad de servicio, entonces el core vuelve a rehabilitarse cuando dicho proceso finalice su timeslice o abandone la CPU.

En el primer caso de reactivación, el incremento detectado en la tasa de fallos de cache respondía muy probablemente al comportamiento natural del programa. Este tipo de situaciones representan "falsas alarmas" para el planificador, y pueden asociarse con fragmentos del programa que muestren un peor comportamiento con la jerarquía cache que el registrado anteriormente. Sin embargo, estas situaciones excepcionales de desactivación sólo se producen durante la transición entre distintas fases de ejecución del programa, ya que el valor promedio que rige el intervalo de calidad de servicio aumenta para ajustarse dinámicamente al nuevo nivel de calidad de servicio (QoS) requerido por el programa con respecto al uso de cache de L2.

En el segundo caso, se registran aquellas situaciones en las que la subida de la tasa de fallos de cache de L2 del proceso prioritario estaba causada directamente por un uso intensivo de cache por otro proceso menos prioritario. En este caso, la situación de conflicto se resuelve a favor del proceso más prioritario, permitiendo una mejora en el rendimiento de dicho proceso con respecto al comportamiento del planificador de Linux 2.6.21.

5.2. Componentes del planificador simbiótico

En esta sección se detallan los componentes principales del planificador simbiótico, así como pequeños detalles de implementación de los mismos sobre Linux 2.6.21. El desarrollo de cada componente se ha llevado a cabo en el lenguaje C, exceptuando la implementación -en ensamblador privilegiado de x86 (Intel)- de las rutinas de acceso a los contadores hardware de monitorización del rendimiento del procesador Intel Core 2 Duo.

El planificador simbiótico está formado por tres componentes

- Herramienta de monitorización
- Motor de activación/desactivación
- Interfaz de configuración

Las siguientes secciones muestran los detalles de cada componente.

5.2.1. Herramienta de monitorización del rendimiento

La herramienta de monitorización es la encargada de proveer al motor de activación/desactivación -componente principal del planificador- de los valores de los parámetros de rendimiento -calculados a partir de los eventos hardware- que le permiten detectar las situaciones de conflicto por compartición de recursos. Se trata del componente de más bajo nivel del sistema, ya que debe acceder a los registros MSR de la máquina (sección A.3.1), que le permiten configurar la unidad de monitorización del rendimiento (PMU).

Esta herramienta sigue el modelo de sistema integrado en planificador- descrito en la sección A.4.3-. El planificador activa el procedimiento de monitorización en las siguientes situaciones:

- Cada cambio de contexto: La activación del componente durante un cambio contexto resulta necesaria para obtener las muestras capturadas por los contadores hardware antes de que otro proceso entre a ejecutar en la CPU. Además de obtener dichas muestras, es necesario resetear y reprogramar los contadores hardware. Gracias a este mecanismo, siempre es posible separar las muestras (de los contadores hardware) de dos procesos que ejecuten en el sistema- a diferencia de los sistemas implementados como drivers-.
- Cada n ticks de planificador (siendo n un parámetro configurable): El planificador simbiótico no sólo desea disponer de la información de monitorización cuando el proceso abandona la CPU, sino mientras este se encuentra en ejecución. Para ello, la herramienta de monitorización toma las muestras de los contadores hardware cada vez que un proceso ejecuta durante n ticks consecutivos. Al disponer de esta información, el planificador puede detectar las situaciones de conflicto por compartición de recursos, antes de que el proceso abandone la CPU.

La herramienta consta de las siguientes capas, organizadas de menor a mayor nivel de abstracción:

- Capa de acceso a PMCs
- Capa de experimentos de monitorización
- Capa de extracción de información de profiling

A pesar de su dependencia de la máquina, el diseño multicapa del componente permite ser adaptado a distintas arquitecturas reescribiendo únicamente la capa inferior. La funcionalidad de esta capa (acceso a PMCs) es ofrecer un interfaz, independiente del mecanismo de gestión de los contadores hardware de cualquier arquitectura, para uso del sistema operativo. Por otra parte la capa de experimentos de monitorización, permite definir experimentos en cada CPU de la máquina, para obtener parámetros de rendimiento (como el IPC o la tasa de fallos de cache) a partir de los valores registrados por los contadores hardware.

De manera adicional, la herramienta permite mostrar al usuario la información de monitorización capturada. La funcionalidad necesaria para dar soporte a esta característica se implementa en la capa de extracción de información de profiling. Cada procesador tiene asociado un buffer circular de gran longitud, donde se almacenan las muestras capturadas con la información adicional para que las herramientas de usuario puedan asociar cada muestra con el proceso generador. La información de cada entrada del buffer es la siguiente:

1. CPU: Es la CPU donde se realizó la cuenta del evento
2. Nombre del evento monitorizado: Secuencia de caracteres asignada por el usuario que describen el evento
3. Valor del evento: Valor obtenido a partir de las medidas efectuadas por los PMCs
4. Identificador de muestra (número de secuencia): Para cada proceso existe un contador que se incrementa cada vez que se procede a la captura de las muestras de los contadores hardware, con independencia de la CPU donde se ejecute. El identificador de muestra permite distinguir el orden en el que las muestras se capturaron y se almacenaron en el buffer. Esto es necesario para que las herramientas de usuario procesen correctamente la información de profiling, ya que una tarea puede migrar de una CPU a otra durante su ejecución.

5. Identificador del proceso PID
6. Identificador del grupo de procesos
7. Contador de programa asociado a la muestra: El contador de programa resulta útil a las herramientas de usuario para asociar las muestras con las secciones de código del ejecutable donde fueron generadas.

5.2.2. Motor de activación/desactivación

El motor de activación/desactivación es el componente principal del planificador simbiótico. Es el encargado de la monitorización de las situaciones de compartición de recursos y e implementa la lógica de desactivación/reactivación de core.

El planificador simbiótico exporta la entrada `/proc/multicore/status` para ser configurado en cualquiera de los siguientes modos de operación:

- Automático: Este es el comportamiento natural del sistema. En este modo su política de calidad de servicio está habilitada y, durante el mismo, el planificador simbiótico alternará la ejecución con un solo core o ambos activados.
- Core deshabilitado: El core 1 permanece completamente deshabilitado y el planificador sirve tareas únicamente en el core 0. La política de calidad de servicio permanece deshabilitada.
- Cores habilitados: Como en el modo previo, la política de calidad de servicio del planificador permanece deshabilitada. Éste es el estado natural del planificador de Linux -con ambos cores activados-; permitiéndose, en todo caso, la ejecución simultánea de dos tareas de distinta prioridad en dos cores del mismo chip.

Este componente, consta de un registro de estado, compartido por ambos cores, que almacena los datos necesarios para implementar la política de calidad de servicio del planificador simbiótico. El registro de estado almacena la siguiente información:

- Estado del planificador simbiótico
- Parámetros de configuración
- Información asociada a situaciones de desactivación

- Contador de ticks transcurridos tras desactivación
- Pid de la tarea prioritaria causante de la desactivación
- Ultimo valor medio de la tasa de fallos de L2 –necesario para determinar el intervalo de calidad de servicio –.

El planificador simbiótico puede permanecer durante su ejecución en los siguientes estados:

- `FORCE_ENABLED`: En este caso, el planificador no realiza ninguna desactivación ya que el planificador permanece en el modo de operación core deshabilitado. Ambos cores permanecen activados y no se lleva a cabo ningún procesamiento de calidad de servicio.
- `FORCE_DISABLED`: En este estado, el core 0 permanece activado y el core 1 desactivado (modo de operación core deshabilitado). El planificador tampoco ejecuta ningún procesamiento de calidad de servicio.
- `AUTO_ENABLED`: Este es el estado más habitual del planificador simbiótico (modo de operación automático). Mientras el planificador se encuentra en este estado, ambos cores están activados, así como su política de calidad de servicio. Desde este estado, el planificador puede efectuar transiciones -en modo de operación automático- al estado `AUTO_DISABLED` cuando desactiva un core.
- `AUTO_DISABLED`: En este estado, uno de los cores permanece desactivado y el proceso prioritario aún no ha alcanzado la tasa de fallos local de L2 que llevaría al planificador a un estado `WAITING_TIMESLICE`. La política de calidad de servicio permanece activada por estar en modo de operación automático.
- `WAITING_TIMESLICE`: Cuando el planificador está en este estado, espera a que el proceso prioritario finalice su timeslice o abandone la CPU para llevar a cabo la reactivación del core desactivado.

5.2.3. Interfaz de configuración

El planificador simbiótico posee un conjunto de parámetros configurables que determinan aspectos esenciales de su modo de operación. El usuario debe ser capaz de establecer el valor de estos parámetros para ajustar el grado de calidad de servicio que desee.

Los parámetros configurables del planificador simbiótico son los siguientes:

- `min_ticks_out`: determina el mínimo número de ticks de sistema que el core permanecerá desactivado
- `max_ticks_out`: establece el número máximo de ticks de sistema que el core permanecerá desactivado.
- `ticks_per_count`: Determina el número de ticks de planificador transcurridos entre dos capturas consecutivas de eventos de los contadores hardware.
- `ticks_start_count`: Hace referencia al número de ticks de sistema, transcurridos entre la creación de un proceso y la primera lectura. Este parámetro se utiliza habiltualmente para ignorar los fallos iniciales de cache que podrían provocar situaciones innecesarias de desactivación de un core.
- `stats_buffer_size`: Determina el número de muestras de almacenadas en el buffer circular de muestras, que el planificador emplea para el calculo del valor promedio o predicción de la tasa local de fallos de cache de L2.
- `l2-miss-rate_threshold`: Determina el umbral de fallos de cache que se emplea en la detección de las situaciones de conflicto por compartición de recursos.

Para permitir la modificación por parte del usuario de los parámetros configurables arriba mencionados, así como para proceder a la obtención del valor actual de estos parámetros; el planificador simbiótico exporta la entrada /proc/multicore/config.

Sin embargo, los parámetros de configuración enumerados previamente no constituyen el único aspecto configurable del sistema; ya que también es posible configurar otros aspectos relacionados con la herramienta de monitorización. Este es el caso de los eventos a monitorizar y la selección de aquellas tareas que exportarán al usuario la información de profiling.

Configuración de los experimentos de monitorización

La herramienta de profiling acepta la configuración de experimentos de monitorización. Para ofrecer esta capacidad al usuario, el planificador simbiótico exporta la llamada al sistema `add_experiments()`, que recibe como parámetro un puntero a una estructura (de tipo `mc_experiment_t`). Esta estructura

permite especificar la configuración de experimentos de monitorización a distintos niveles y esta constituida por múltiples arrays (uno por procesador) con conjuntos de eventos para ser monitorizados en cada CPU. Cada array está compuesto por los siguientes elementos:

- Configuración de bajo nivel para configurar los contadores hardware (PMCs) en la CPU destino
- Especificación de los parámetros de rendimiento a calcular a partir de la información extraída de los contadores hardware
- Umbrales de calidad de servicio para la política de planificación

Configuración de bajo nivel

Esta información permite especificar la configuración de los contadores hardware (PMCs) en la CPU destino. Los *eventos hardware* requieren distintos mecanismos (implementaciones) para llevar a cabo los procedimientos de configuración, reseteo y cuenta de los contadores hardware[2].

Sin embargo, la herramienta de monitorización del rendimiento ofrece un tipo de datos para abstraer al sistema operativo de los distintos mecanismos que soporta la arquitectura para configurar los eventos *eventos hardware* . Esta abstracción se denomina *evento de bajo nivel*.

Especificación de los parámetros de rendimiento

Los eventos de bajo nivel ofrecen una abstracción de los distintos procedimientos de interacción de distintos tipos de eventos hardware. Sin embargo, estos eventos son de bajo nivel de abstracción, ya que no permiten especificar parámetros de rendimiento –como la tasa de fallos de cache o el número de instrucciones por ciclo (IPC)–.

Este hecho, motiva la introducción de los eventos de alto nivel que permiten representar parámetros de rendimiento mediante distintas combinaciones de operaciones cuyos operandos son los valores asociados a eventos de bajo nivel. Gracias a los eventos de alto nivel, el planificador simbiótico puede acceder directamente a parámetros de rendimiento que facilitan la implementación de la política de calidad de servicio.

[2]El procesador Intel Core 2 Duo soporta dos modos de configuración diferentes: eventos configurables y eventos de cuenta fija (sección A.3.3)

Umbrales de calidad de servicio para la política de planificación

Asociado a cada parámetro de rendimiento, puede definirse aquel conjunto de valores del mismo que pueden representar situaciones de conflicto -en las que un proceso prioritario compite por el uso de recursos con procesos menos prioritarios-. Los intervalos que contienen a estos valores, se definen mediante un umbral de calidad de servicio. El valor promedio (predicción) para un parámetro de rendimiento determina el centro c del intervalo en cada instante de muestreo t; y el umbral fijado ε determina la amplitud del intervalo de modo que:

$$\boxed{Intervalo(t) = [c(t) - \varepsilon, c(t) + \varepsilon]}$$

El nuevo valor del evento de alto nivel (parámetro de rendimiento) obtenido en cada momento, se comparará con el intervalo de calidad de servicio y con el valor anterior registrado para dicho parámetro, pudiendo distinguirse los siguientes casos.

1. Retornos al intervalo desde un valor inferior
2. Retornos al intervalo desde un valor superior
3. Salidas del intervalo hacia un valor inferior
4. Salidas del intervalo hacia un valor superior

La configuración especificada por la estructura pasada como parámetro a `add_experiments()`, permite fijar acciones de activación y desactivación de un core asociadas los cuatro tipo de situaciones mencionadas anteriormente. En La configuración por defecto del planificador simbiótico, el parámetro de la tasa de fallos local de L2 provoca acciones de desactivación asociadas a las salidas del intervalo hacia valores superiores, y acciones de reactivación asociadas con retornos al intervalo desde un valor superior.

Procedimiento de extracción de las muestras de monitorización de los procesos

Para acceder a la información sobre las muestras capturadas por la herramienta de monitorización, el usuario debe realizar las siguientes acciones:

1. Introducir en el núcleo la configuración de los eventos a monitorizar mediante la llamada al sistema `add_experiments()`.

2. Seleccionar el proceso o procesos que han de volcar la información de monitorización al buffer de muestras asociado a cada CPU. Esto puede realizarse de las siguientes formas:

   a) Desde el shell,

      1) Invocando la orden:
      `$ echo $PID > /proc/multicore/statisticks`
      Donde $PID almacena el identificador del proceso que deseamos monitorizar
      2) También puede emplearse el siguiente script (`launch.sh`) para lanzar cualquier programa

```
#!/bin/sh
programa=$*
pid=$$
echo $PID > /proc/multicore/statisticks
#Ejecucion del programa
exec $programa
```

      e invocando este script desde el shell:

```
$ sh launch.sh /home/usuario/bin/mi_programa
```

   b) Desde el código de un programa lanzador auxiliar (en C o Perl), que presente la siguiente estructura:

      1) Crear un proceso mediante fork()
      2) En el código del nuevo proceso hijo
         - escribir `self` en la entrada /proc/multicore/statisticks
         - Invocar exec() para ejecutar el código del programa que que se desee monitorizar
      3) Invocar exit() para salir en el código del proceso padre.

3. Procesar la entrada /proc/multicore/statistics cada cierto tiempo para acceder a la información de profiling. Cada operación de lectura a dicha entrada permite únicamente al acceso a un máximo de 4KB de información de profiling. Esto es así, porque el núcleo reserva un buffer de 4KB por cada fichero /proc.

6.2.3.2 Entradas en /proc

El planificador simbiótico exporta las siguientes entradas /proc para permitir la interacción con el usuario:

- /proc/multicore/status:
  - Operaciones de escritura: Permiten establecer el modo de operación del planificador. Aceptan tres cadenas como entrada:
    - `auto`: Modo de operación automático
    - `disable`:Modo de operación de core 1 desactivado
    - `enable`: Modo de operación de ambos cores activados
  - Operaciones de lectura: Muestran el estado actual en el que se encuentra el planificador:
    - `auto_enabled`
    - `auto_disabled`
    - `force_enabled`
    - `force_disabled`
    - `waiting_timeslice`
- /proc/multicore/config: Esta entrada permite alterar y consultar el valor de los parámetros de configuración enumerados en la sección anterior (exceptuando `L2-miss-rate_threshold`).
  - Operaciones de escritura: Aceptan como entrada el nombre del parámetro configurable seguido del valor entero positivo que se desea asignar.
  - Operaciones de lectura: Muestran el valor de todos los parámetros de configuración.
  - /proc/multicore/experiments: Mediante las operaciones de escritura, la entrada puede ser configurada para realizar distintas acciones cuando se realice una acción posterior de lectura sobre ella. Durante una operación de escritura, la entrada acepta las siguientes opciones de configuración:
    - `print_experiments`: Imprime el número de eventos de alto y bajo nivel del sistema.
    - `reset_statisticks`: Permite borrar las estadísticas de monitorización asociadas a todos los procesos del sistema
    - `clear_experiments`: Borra la configuración insertada mediante la llamada al sistema `add_experiments()` y resetea las estadisticas de monitorización almacenadas en cada descriptor de proceso
    - `unset_profiling`: Deshabilita el volcado de muestras de cualquier proceso que tuviera habilitada esta característica

- `show_all`: Muestra los nombres simbólicos de eventos de alto y bajo nivel.
- `show_hl`: Muestra los nombres simbólicos de los eventos de alto nivel
- `show_ll`: Muestra los nombres simbólicos de los eventos de alto nivel
- `show_buffer_size`: Muestra el número de muestras almacenados en los buffers de profilings asociados a todas las CPUs Un ejemplo de uso de esta entrada (borrado de los experimentos), es el siguiente:

```
$ echo clear_experiments > /proc/multicore/experiments
$ cat /proc/multicore/experiments
```

- /proc/multicore/statistics: Entrada creada para ofrecer al usuario la información de monitorización capturada por los contadores hardware: o
  - Operaciones de escritura: La entrada acepta distintos valores de entrada:
    - `enable`: Habilita globalmente el volcado de información al buffer de profiling de cada CPU. El volcado solo se lleva a cabo por los procesos que han sido seleccionados explícitamente.
    - `disable`: Deshabilita globalmente el volcado de información a los buffers de profiling
    - `self`: Habilita al proceso invocador de la operación de escritura para exportar su información de profiling al usuario
    - `noself`: Deshabilita la característica anterior
    - Identificador de un proceso: Si se introduce un PID válido, aquella tarea con dicho PID mostrará la información de profiling desde ese preciso instante
  - Operaciones de lectura: Permiten el acceso, a la información de profiling (hasta 4KB por cada operación de lectura), construida a partir de las muestras de los procesos almacenadas en los buffers de profiling de cada CPU. Dicha información se representa en modo texto por columnas. Cada columna representa un campo de cada muestra almacenada (sección 5.2.1).

Capítulo 6

Resultados

Para mostrar las prestaciones del *planificador simbiótico* es necesario reproducir las situaciones de conflicto por compartición del segundo nivel de cache que transcurren durante la ejecución simultánea de dos tareas de distinta prioridad ,una en cada core. De este modo, el uso de benchmarks que presenten un uso intensivo de la jerarquía cache permitirá aumentar la probabilidad de aparición de este tipo de situaciones de conflicto.

Por otra parte, cabe destacar que el objetivo último del planificador simbiótico es dar soporte a una política de calidad de servicio que optimice el rendimiento de los procesos más prioritarios (speedup) sin degradar drásticamente la productividad global del sistema. Para medir la productividad del sistema es necesario introducir el concepto de factor de productividad (llamado *system speedup* en [18]) empleado frecuentemente para contabilizar la calidad de la simbiosis asociada a la ejecución simultánea de dos programas en distintas CPUs lógicas que compartan recursos del procesador o de la jerarquía cache.

Definición 3 *Dados dos programas independientes P_0 y P_1 . Sea $t_{solo}(P_i)$ el tiempo de ejecución del proceso P_i cuando ejecuta con todos los recursos diponibles (solo) y sea $t_{par}(P_i, P_j)$ el tiempo de ejecución del proceso P_i cuando ejecuta simultáneamente en el sistema con el proceso P_j. Se define el factor de productividad como:*

$$\text{factorProductividad}(P_0, P_1) = 100(\frac{t_{solo}(P_0)}{t_{par}(P_0, P_1)} + \frac{t_{solo}(P_1)}{t_{par}(P_1, P_0)} - 1)$$

Este parámetro viene representado como un porcentaje y, en función de valores obtenidos puede deducirse el grado de contención por compartición de recursos en arquitecturas CMP o SMT:

- 100 %: Este valor indica una simbiosis perfecta, es decir, la existencia de recursos compartidos no ha afectado negativamente a la ejecución, ya que los procesos han ejecutado en el mismo tiempo que si dispusieran de todos los recursos compartidos a su disposición.
- Valores positivos: Cuanto mayor es este valor, mejor es la calidad de la simbiosis y por lo tanto la contención por compartición de recursos es menor.
- Valores negativos: La presencia de valores negativos se produce en situaciones en las que el tiempo de ejecución en paralelo de ambos procesos es mayor que la suma de sus tiempos de ejecución al ejecutar secuencialmente. Estos valores muestran una situación de contención por compartición de recursos muy severa.

La primera parte de este capítulo describe el procedimiento de selección de los benchmarks candidatos para ejecutar, tomados del conjunto de benchmarks de SPEC CPU2006. En la segunda parte se describe el proceso de obtención del valor de los parámetros de configuración que garantizan un mejor comportamiento de la política de calidad de servicio. Finalmente, en la última parte se analizan los resultados obtenidos durante la ejecución de pares de benchmarks seleccionados.

6.1. Caracterización de SPEC CPU 2006

El 24 de agosto de 2006, la corporación SPEC (Standard Performance Evaluation Corporation) anunció el lanzamiento de CPU2006, la última generación de benchmarks intensivos en CPU que constituye un estándar de facto para la industria [19]. Por otra parte, el conjunto de benchmarks de SPEC CPU también se ha convertido en el más utilizado en la investigación en el campo de la arquitectura de computadores.

El conjunto de benchmarks de SPEC CPU está constituido por programas enteros y de punto flotante intensivos en CPU para medir el impacto en el rendimiento de los procesadores, la memoria y los compiladores. Para mantenerse al mismo nivel que los avances tecnológicos, las mejoras de los compiladores y las nuevas cargas de trabajo de las distintas configuraciones, la corporación SPEC lanza nuevas versiones de SPEC CPU en las que se añaden nuevos programas, se eliminan programas que son susceptibles de optimizaciones injustas del compilador, se incrementa el tiempo de ejecución de cada

programa y se incrementa la intensidad de los accesos a memoria de cada benchmark.

En [20] se muestran las amplias diferencias existentes entre las versiones de SPEC CPU 2000 y SPEC CPU 2006 en cuanto a uso de memoria y tiempo de ejecución. Las tablas 6.1 y 6.2 muestran la descripción de los benchmarks de la suite SPEC CPU 2006.

Nombre	Lenguaje	Descripción
400.perlbench	C	PERL Programming Language
401.bzip2	C	Utilidad de compresión
403.gcc	C	Compilador de C
429.mcf	C	Optimización combinatoria
445.gobmk	C	Juego de inteligencia artificial: GO
456.hmmer	C	Búsqueda de secuencias de genes
458.sjeng	C	Juego de inteligencia artificial: ajedrez
462.libquantum	C	Física cuantica
464.h264ref	C	Compresión de vídeo
471.omnetpp	C++	Simulación de eventos discretos
473.astar	C++	Algoritmos de búsqueda (grafos)
483.xalancbmk	C++	Procesamiento XML

Cuadro 6.1: Benchmarks enteros de SPEC CPU2006)

En [21], se muestran distintos parámetros de rendimiento obtenidos mediante los contadores hardware en una máquina Sun Blade 2000. A partir de esta información se seleccionan aquellos benchmarks de SPEC CPU2006 con mayor número de fallos de cache de segundo nivel por cada mil instrucciones. Esta medida se utiliza para realizar un primer filtrado de aquellos benchmarks que presentan un uso intensivo de cache. Los benchmarks resultantes son: GemsFDTD, astar, gcc, h264ref, lbm, leslie3d, libquantum, milc, omnetpp y soplex. Posteriormente, mediante la herramienta de monitorización del rendimiento del planificador simbiótico , se calculan los distintos valores que presenta cada benchmark a lo largo del tiempo para el número de instrucciones por ciclo y la tasa de fallos local de L2. La herramienta de monitorización, que hace uso de los contadores hardware de la máquina (Intel Core 2 Duo E6300), se configura para leer las muestras de los contadores en cada cambio de contexto entre procesos. El objetivo de esta monitorización, es capturar el comportamiento natural del programa con respecto a jerarquía cache cuando dispone del 100 % de la cache de segundo nivel para su uso, sin compartirla con otro proceso que ejecute en el core opuesto. Para reproducir este comportamiento el sistema queda establecido en un modo de baja carga

(runlevel 1). Cabe destacar que para algunos benchmarks, SPEC realiza distintas ejecuciones con diferentes datos de entrada (data input sets). De este, modo la notación #i junto al nombre del benchmark se emplea para indicar el conjunto de datos entrada utilizado en la ejecución.

6.1.1. Características de fallos de cache de segundo nivel e instrucciones por ciclo

A continuación se muestran las características de fallos de cache de segundo nivel e instrucciones por ciclo asociadas la ejecución de distintos benchmarks con distintos conjuntos de datos de entrada. Estas características se han obtenido a partir de la información extraida por la herramienta de monitorización del rendimiento del planificador simbiótico.

Nombre	Lenguaje	Descripción
410.bwaves	Fortran	Dinámica de fluidos
416.gamess	Fortran	Química Quántica
433.milc	C	Física: Cromo-dinámica cuántica
434.zeusmp	Fortran	Physics CFD
435.gromacs	CFortran	Bioquímica / Dinámica molecular
436.cactusADM	CFortran	Física / Relatividad general
437.leslie3d	Fortran	Dinámica de fluidos
444.namd	C++	Biología molecular
447.dealII	C++	Análisis de elementos finitos
450.soplex	C++	Programación lineal, Optimización
453.povray	C++	Ray-tracing sobre imágenes
454.calculix	CFortran	Mecánica estructural
459.GemsFDTD	Fortran	Electromagnetismo computacional
465.tonto	Fortran	Química Cuántica
470.lbm	C	Dinámica de fluidos
481.wrf	CFortran	Predicción del tiempo
482.sphinx3	C	Reconocimiento de voz

Cuadro 6.2: Benchmarks de punto flotante de SPEC CPU2006)

GemsFDTD

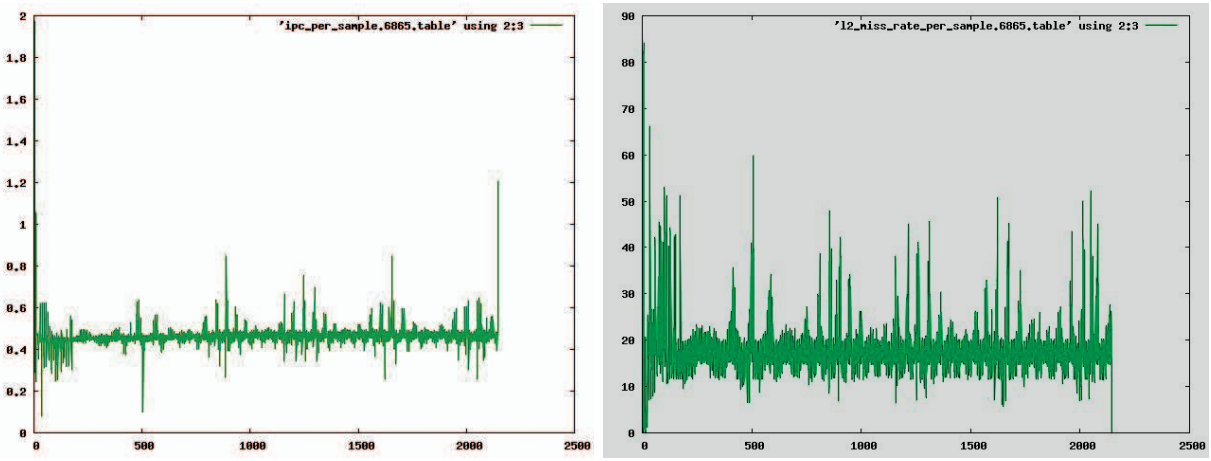

Figura 6.1: GemsFDTD: Graficas de IPC y tasa de fallos local de L2

astar

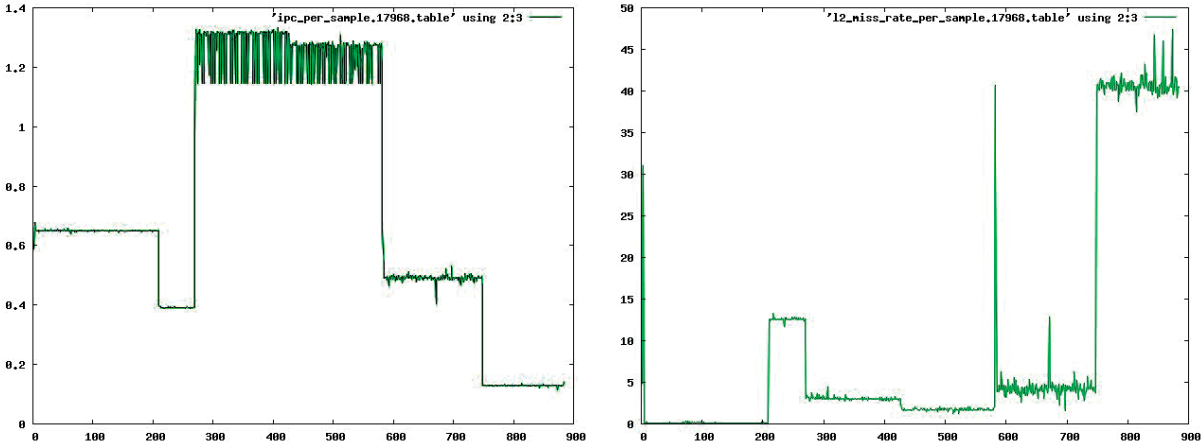

Figura 6.2: astar: Graficas de IPC y tasa de fallos local de L2

gcc

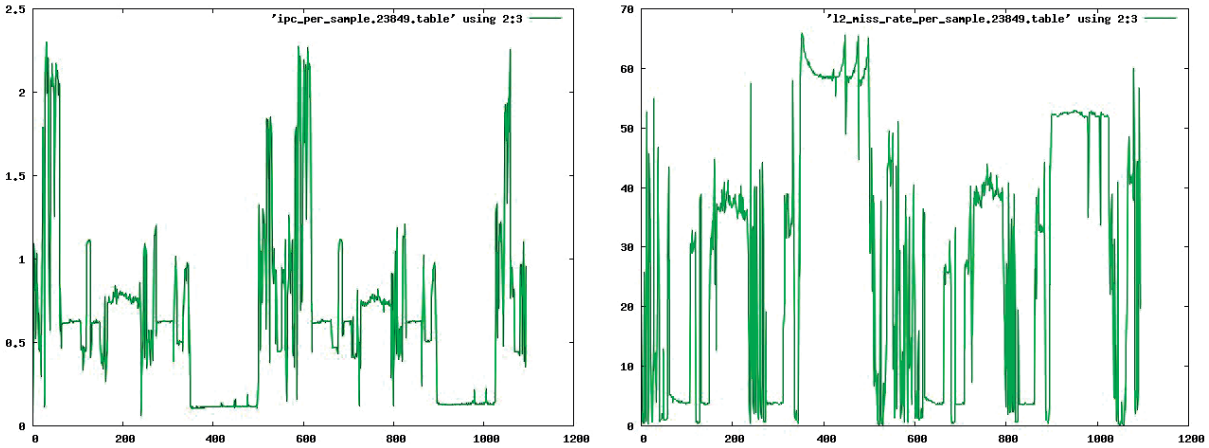

Figura 6.3: gcc: Graficas de IPC y tasa de fallos local de L2

h264ref

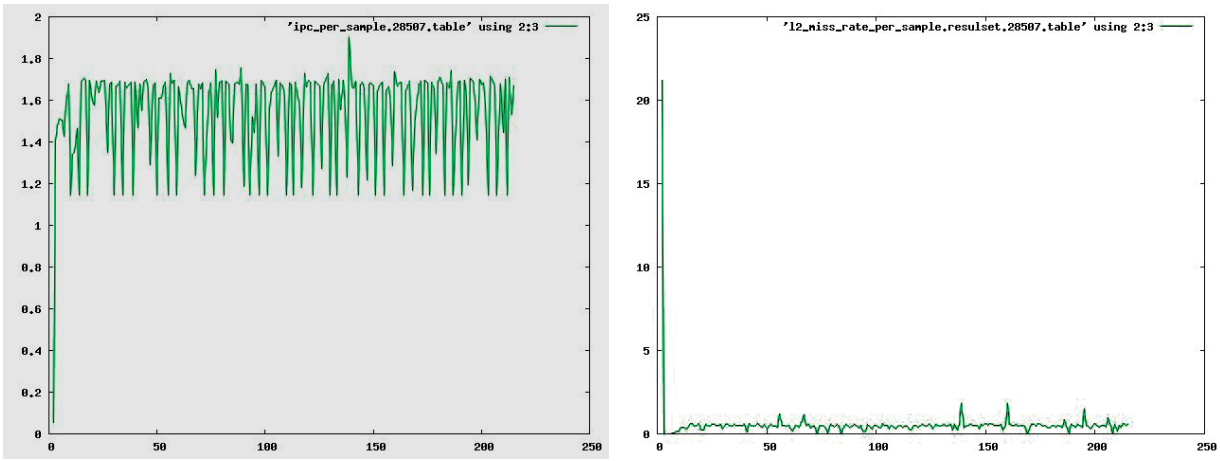

Figura 6.4: h264ref: Graficas de IPC y tasa de fallos local de L2

lbm

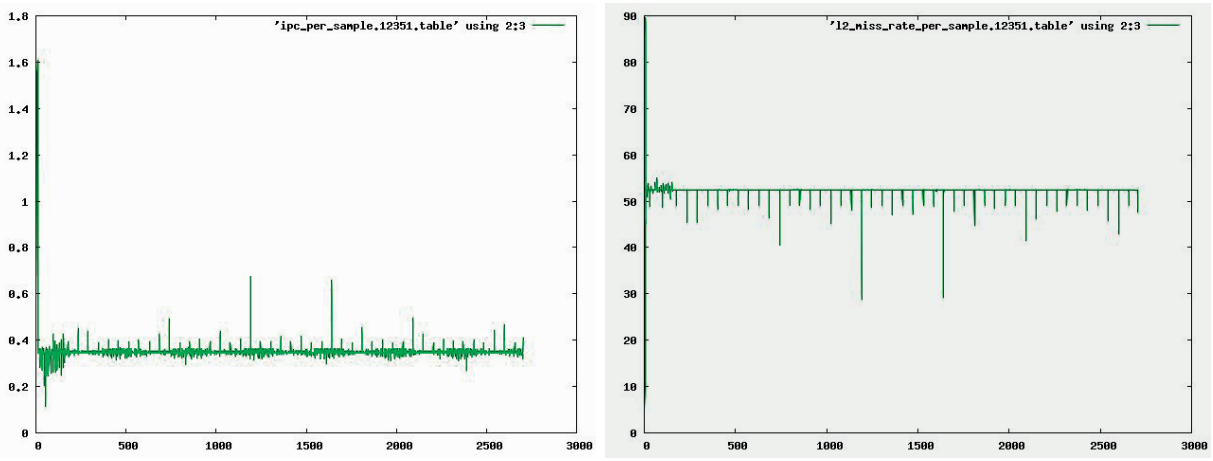

Figura 6.5: lbm: Graficas de IPC y tasa de fallos local de L2

leslie3d

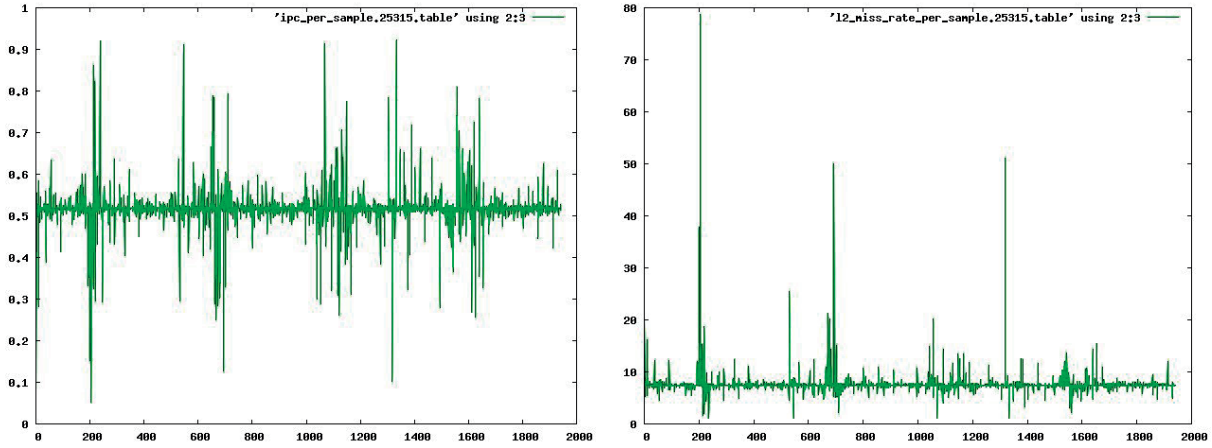

Figura 6.6: leslie3d: Graficas de IPC y tasa de fallos local de L2

milc

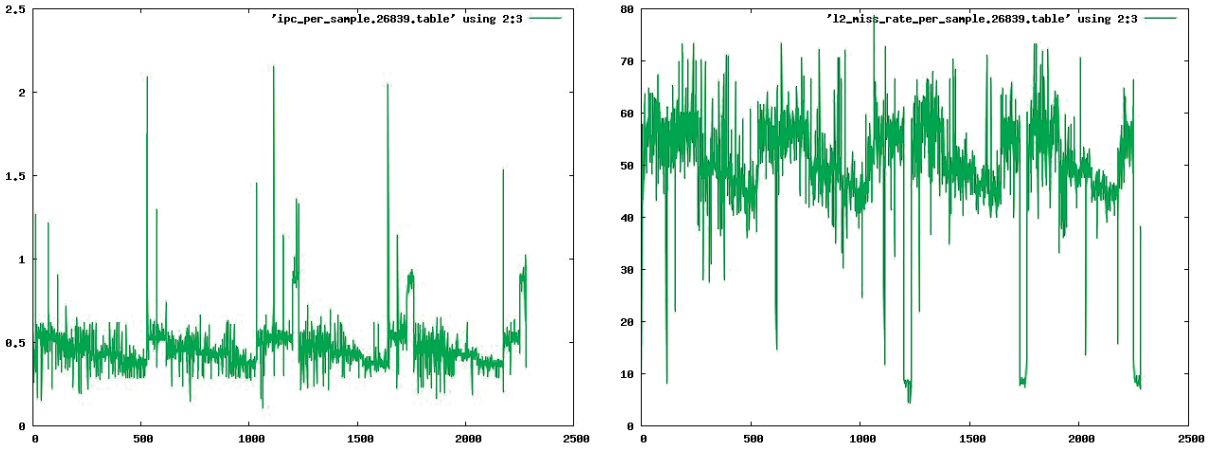

Figura 6.7: milc: Graficas de IPC y tasa de fallos local de L2

omnetpp

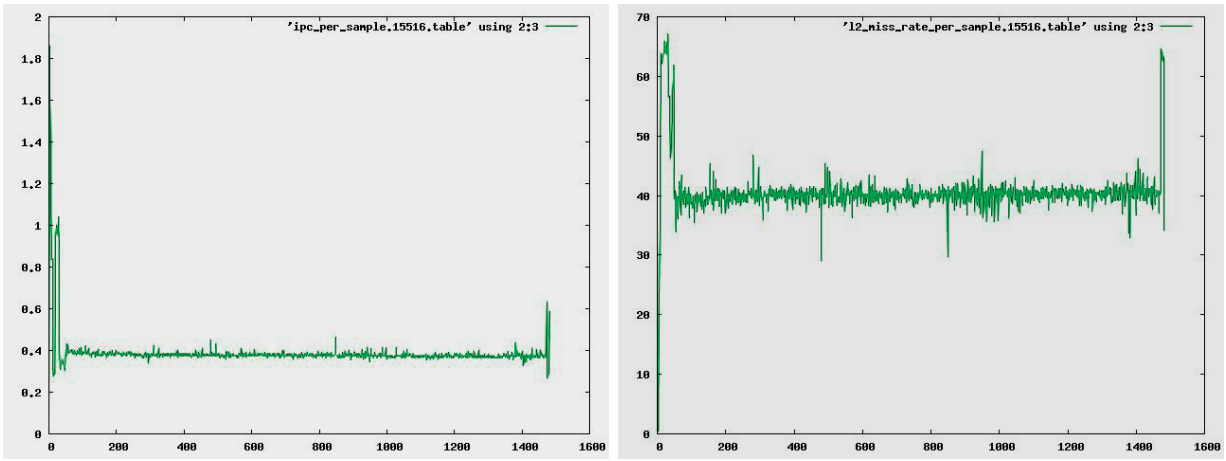

Figura 6.8: omnetpp: Graficas de IPC y tasa de fallos local de L2

soplex

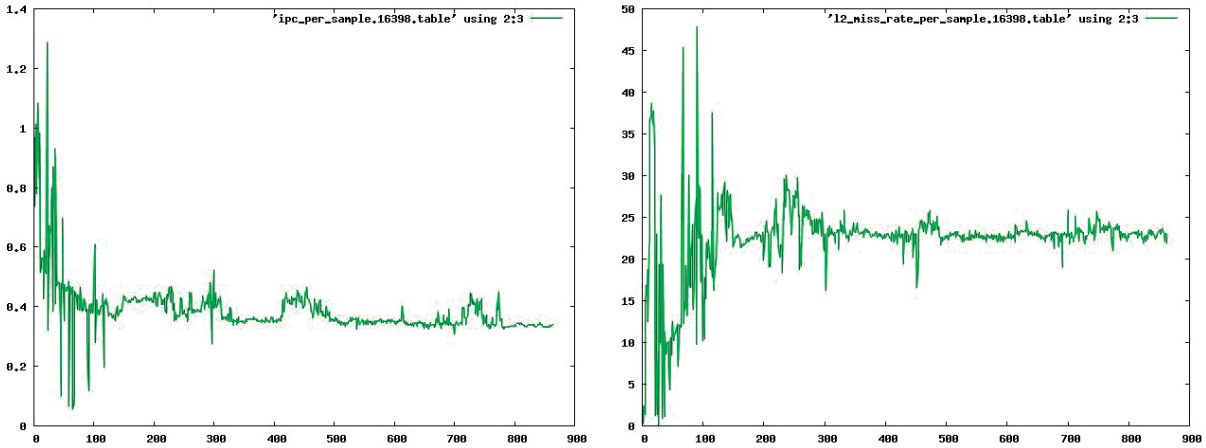

Figura 6.9: soplex: Graficas de IPC y tasa de fallos local de L2

6.1.2. Selección de benchmarks candidatos

La figura 6.10, muestra el valor medio de la tasa de fallos local de cache de L2 que presenta cada una de las anteriores ejecuciones. Para poner a prueba la política de calidad de servicio del planificador simbiótico, se escoge un subconjunto de las ejecuciones de aquellos benchmarks y datos de entrada que presenten un uso más intensivo de la cache de segundo nivel. Esta selección permite aumentar la probabilidad de aparición de las situaciones de conflicto por compartición.

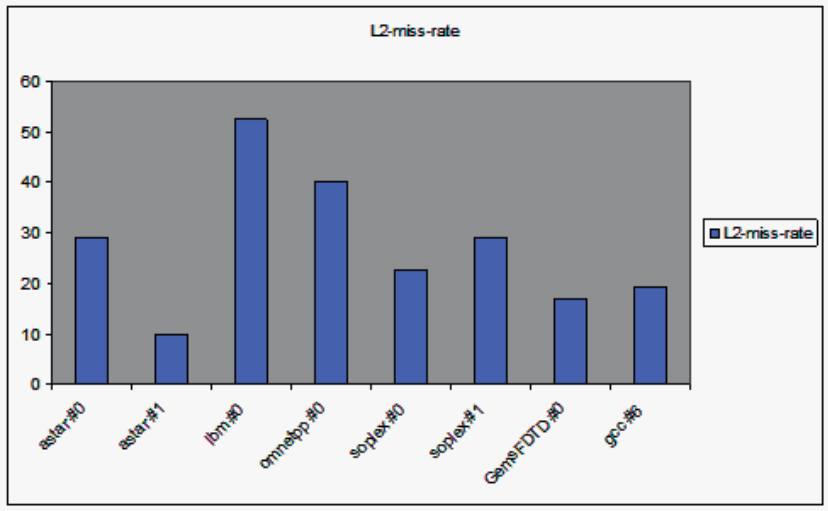

Figura 6.10: Tasa de fallos local de L2 de benchmarks candidatos

6.1.3. Exploración de parámetros de configuración

Como se describe en la sección 5.2.3, el planificador simbiótico dispone de distintos parámetros de configuración que le permiten ajustar el nivel de calidad de servicio deseado, ya que el speedup de los procesos prioritarios y productividad varían para distintos conjuntos de valores para los parámetros de configuración.En este trabajo no hemos considerado una exploración exhaustiva de dichos parámetros. Como primera aproximación, nos hemos centrado en estudiar:

- `min_ticks_out` y `max_ticks_out` (intervalo de desactivacion)
- `ticks_per_count`
- `L2-miss-rate_threshold`

Para realizar un análisis de los valores óptimos, se escoge un conjunto finito de valores para cada parámetro de configuración del planificador simbiótico y se procede a realizar múltiples ejecuciones de exploración que permitan encontrar los conjuntos de valores que presenten mejores resultados. El benchmark de referencia seleccionado para llevar a cabo dicho análisis es el astar con el primer conjunto de datos de entrada (astar #0). Antes de activar la política de calidad de servicio del planificador es necesario tomar los tiempos de ejecución del programa relativa a dos tipos de ejecución:

- Tiempo de ejecución del proceso cuando ejecuta solo en el sistema. En este caso el proceso dispone de toda la cache de L2 disponible.
- Tiempos de ejecución del mismo programa ejecutado simultáneamente en distintos cores y con distinta prioridad. De este modo, se obtendrá el tiempo de ejecución del proceso más prioritario (tprio) y el del proceso menos prioritario (`tnon_prio`).

El experimento de exploración, consiste en la ejecución simultánea de dos instancias de distinta prioridad del programa astar #0. Se realiza una ejecución con la política de calidad de servicio activada para cada conjunto de valores posibles asignados (conjunto finito) a los parámetros de configuración.

Los primeros experimentos, se destinaron a la elección de los valores óptimos para el *intervalo de desactivación* (determinado por los parámetros de configuración `min_ticks_out` y `max_ticks_out`. Los experimentos muestran que la anchura del intervalo determina:

- El tiempo que el proceso más prioritario dispone para recuperarse de las situaciones de conflicto. Por lo tanto una mayor anchura del intervalo favorece al proceso más prioritario.
- En las situaciones de desactivación provocadas por el comportamiento natural del programa, el intervalo determina el tiempo que el proceso menos prioritario ejecutará en el mismo core que el más prioritario. Por lo tanto, cuanto más ancho sea el intervalo mayor será la degradación del rendimiento del proceso menos prioritario

Por otra parte, el valor del extremo inferior del intervalo (parámetro `min_ticks_out`), no sólo determina el tiempo mínimo que el core permanecerá desactivado, sino que también tiene las siguientes implicaciones en el rendimiento:

- Cuanto mayor sea el extremo inferior del intervalo, mayor será la degradación del rendimiento del proceso menos prioritario; ya que ,durante ese tiempo el proceso menos prioritario ejecutará en el mismo core que el más prioritario.
- Cuanto menor sea el extremo inferior del intervalo, antes se actualizará el valor promedio que determina el intervalo de calidad de servicio (sección 5.1.2). Por lo tanto valores menores del parámetro `min_ticks_out` provocarán una disminución del grado de calidad de servicio del proceso más prioritario.

El valor óptimo, encontrado para distintos intervalos de desactivación es [10,50] [1] . Con este valor se realizan distintas pruebas variando el número de ticks por cuenta (`ticks_per_count`) y el umbral de la tasa de fallos local de cache de L2 (`l2-miss-rate_threshold`). La figura 6.11 muestra el speedup del proceso más prioritario (variando estos parámetros) con respecto al tiempo de ejecución obtenido con la política de planificación desactivada (comportamiento natural de Linux). De manera complementaria, la figura 6.11 muestra el factor de productividad del sistema (definido al inicio del capítulo) obtenido para las distintas ejecuciones.

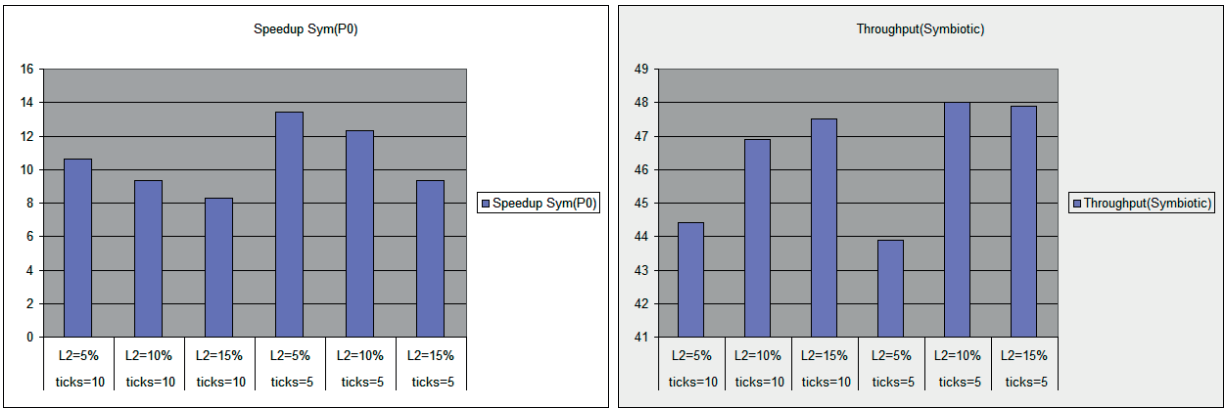

Figura 6.11: Graficas de *speedup* del proceso más prioritario y *factor de productividad* asociado com los experimentos de exploración con la ejecución (astar #0 vs. astar #0)

Los valores óptimos a elegir, para cada parámetro serán aquellos que presenten mayor *speedup* sin degradar la productividad. Como muestran las gráficas asociadas a ambos factores, los valores óptimos para un intervalo de desactivación de [10,50], son los pares (`ticks_per_count`, `l2-miss-rate_threshold`) con valores (5 ticks ,5 %) y (5 ticks,10 %). El primer par de valores consigue el mayor speedup para el proceso más prioritario pero posee un factor de productividad muy bajo. Por otra parte el par (5,10 %) presenta una buena relación speedup-productividad.

[1] La notacion $[a, b]$ para el intervalo de desactivación indica que el extremo inferior del intervalo tiene un valor de a ticks y el extremo superior del intervalo tiene un valor de b ticks

6.1.4. Resultados de calidad de servicio

Gracias al proceso de exploración de los valores de los parámetros de configuración, ha sido posible determinar los valores que optimizan la relación rendimiento-productividad del benchmark de referencia (astar #0). Es por ello que los experimentos de calidad de servicio con los benchmarks seleccionados emplean los siguientes valores:

- `ticks_per_count` = 5 ticks
- intervalo de desactivación [`min_ticks_out`, `max_ticks_out`] = [10,50] (en ticks)
- `l2-miss-rate_threshold` = $\{8\,\%, 10\,\%\}$

La figura 6.12 muestra la accion del planificador simbiótico. En esta gráfica se representa el speedup del proceso más prioritario (eje de ordenadas) con respecto al planificador de Linux - sin calidad de servicio-. En el eje de abscisas se muestran los pares de benchmarks ejecutados simultáneamente durante el experimento, siendo el más prioritario el que aparece en la parte inferior.

Como puede percibirse, el planificador logra un speedup más que aceptable para el proceso más prioritario en el 70 % de los experimentos. Sin embargo, existen algunos benchmarks donde la política de calidad de servicio no consigue obtener mejoras del tiempo de ejecución del proceso más prioritario. No obstante, en estos casos la sobrecarga introducida por el planificador simbiótico es inferior al 2 %.

El máximo *speedup* que puede lograrse para el proceso prioritario viene determinado por el tiempo de sobrecarga introducido por la contención por compartición de recursos entre los cores. Este tiempo es la diferencia entre el tiempo de ejecución de un proceso cuando ejecuta con otro, y el tiempo de ejecución de dicho proceso cuando está solo. La gráfica de la figura 6.13 muestra la relación entre la mejora obtenida y la fracción de mejora asociada al tiempo de sobrecarga. Cabe destacar, que la cache de segundo nivel no es el único recurso compartido entre los cores, ya que el *front side bus* también se encuentra compartido. Es posible, que el impacto por la compartición del *front side bus* determine un porcentaje considerable del tiempo de sobrecarga.

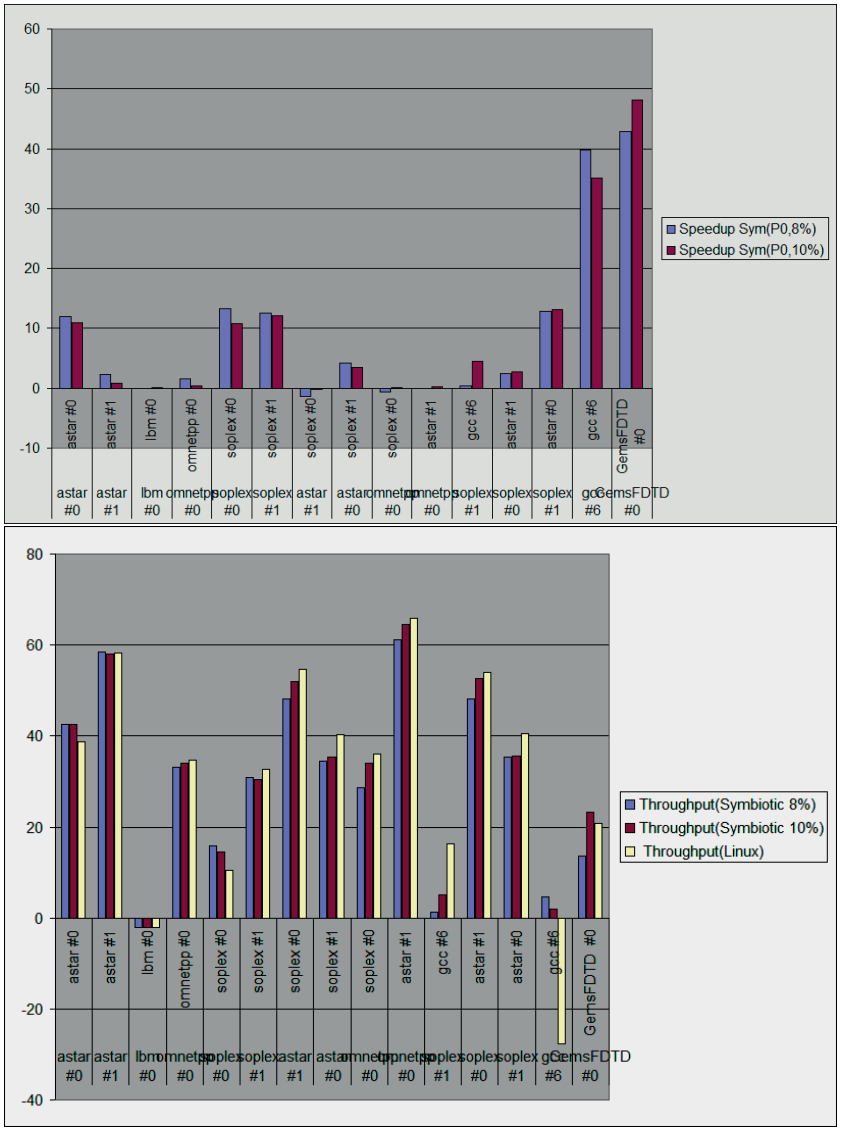

Figura 6.12: Graficas del *speedup* del proceso más prioritario y *factor de productividad* asociado a los experimentos de calidad de servicio con el subconjunto de *benchmarks* con uso intensivo de cache

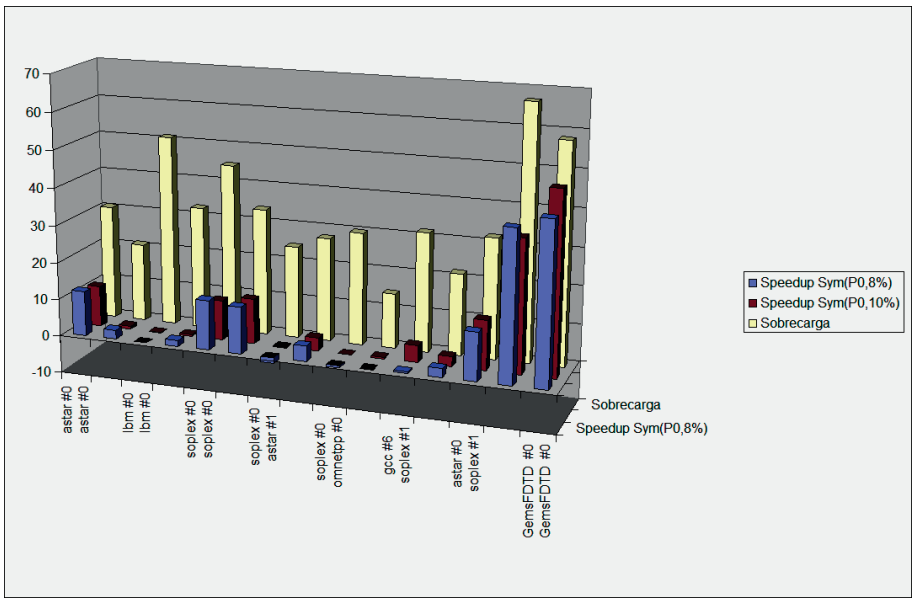

Figura 6.13: Fracción de mejora con respecto al speedup ideal

Por otra parte, como complemento de la información del *speedup*, la figura 6.12, muestra el valor del factor de productividad (definido al inicio del capítulo) para la ejecución con los distintos umbrales de la tasa de fallos local de L2, y para la ejecución del planificador nativo de Linux (sin soporte para calidad de servicio). Puede realizarse la siguiente lectura a partir de los resultados:

- El 92 % de los casos no presentan un impacto muy negativo en la productividad. De este modo, la productividad no se ve degradada para lograr QoS.
- Un valor superior del umbral de fallos de cache permite lograr una mayor productividad en la mayor parte de los casos. Esto es así ya que el establecimiento de un umbral superior, provoca menos situaciones de desactivación de un core y, por tanto el proceso menos prioritario pierde su afinidad a la cache de nivel 1 en menos ocasiones. * En algunas ocasiones, donde la compartición de recursos provoca una severa degradación del factor de productividad (gcc #6 vs. gcc #6), la política de calidad de servicio puede conseguir aliviar el efecto de la contención.

Capítulo 7

Trabajo Relacionado

Bellosa y Steckermeier [22] fueron los primeros en sugerir el uso de los contadores hardware de monitorización del rendimiento para detectar los conflictos de compartición entre threads, ubicándolos en el mismo procesador. Debido a la dificultad de acceso a los contadores hardware en aquel tiempo- hace diez años en un Convex SPP 1000- no pudieron mostrar resultados experimentales de sus teorías.

Bellosa propuso usar la información del TLB para reducir los fallos de cache entre cambios de contexto de los procesos y maximizar la reutilización de las lineas de cache, identificando aquellos threads que comparten las mismas regiones de datos [23]. Los threads que comparten regiones se planifican secuencialmente -uno inmediatamente detrás de otro- para maximizar las oportunidades de reutilización de bloques de cache. Koka y Lipasti perseguían los mismos objetivos y proporcionaron información mas detallada sobre los problemas encontrados al emplear la misma estrategia [24]. Sin embargo, el trabajo de estos dos grupos de investigación se centra en sistemas monoprocesador y no en CMPs.

Muchos investigadores han analizado el problema de la minimización de los fallos de cache de capacidad y conflicto en arquitecturas con un segundo nivel de cache compartido. Snavely y Tullsen publicaron interesantes trabajos en el área de la coplanificación (co-sheduling) que muestran el problema de la planificación convencional y los potenciales beneficios de rendimiento logrados mediante planificación simbiótica en un entorno de simulación [12]. Con el lanzamiento de los procesadores de Intel con tecnología Hyperthreading, Nakajima y Pallipadi analizaron el impacto de la coplanificación en estos sistemas reales [25]. Parekh et al. hicieron uso de los contadores de monitorización del rendimiento para obtener información sobre los fallos de cache y

desplegar una técnica de coplanificación muy sofisticada [26].

McGregor et al. y El-Moursy et al., muestran que emplear únicamente una heurística basada en fallos de cache, resulta insuficiente para determinar las situaciones de conflicto por compartición de recursos en multiprocesadores formados por múltiples chips con SMT. Esto es así, porque los procesadores SMT comparten otros muchos recursos microarquitectónicos además del primer y segundo nivel de cache [27, 28]. El- Moursy et al. demuestran que el empleo de una heurística basada en el número de instrucciones por ciclo permite detectar un mayor número de situaciones de conflicto por compartición de recursos en procesadores SMT.

Suh et al. introducen una aproximación genérica a la planificación *memory-aware*, en la cual los threads se planifican en función de su porcentaje de uso de cache [29, 30]. Por ejemplo, la ejecución en paralelo de un proceso con uso intensivo de cache con otro proceso con bajo índice de utilización de cache constituye una buena decisión de planificación memory-aware. Fedorova et al. analizan el diseño de un planificador simbiótico que emplea una heurística de coplanificación basada en la tasa de fallos de cache, cuyo objetivo es de reducir los fallos de capacidad y conflicto [11, 31].

Otros trabajos analizan distintas estrategias de calidad de servicio (QoS) en arquitecturas SMT y CMP. Iyer et al. analizan el problema de la asignación de los recursos compartidos a cada core en plataformas CMPs presentando distintas políticas de asignación de recursos (estáticas y dinámicas) [32, 33]. Estas políticas tienen en cuenta tanto los requisitos del usuario (prioridades) como las características de uso de recursos de cada programa. Cazorla propone distintas modificaciones microarquitectónicas en los procesadores con SMT para dar soporte hardware de calidad de servicio [34, 35, 36].

En relación con el análisis del comportamiento del conjunto de benchmarks de SPEC CPU2006 con respecto a la jerarquía de memoria, Henning realiza un estudio basado en la utilización de los contadores hardware de monitorización del rendimiento [21]. Por otra parte, el mismo autor en [20] realiza una comparativa entre el consumo de memoria de los benchmarks de SPEC CPU2006 con los benchmarks de la anterior versión de esta misma suite –SPEC CPU2000–.

Finalmente, Siddha [15] enumera las últimas modificaciones incorporadas al kernel Linux para desplegar distintas políticas de equilibrado de carga en servidores constituidos por múltiples chips CMP. En dicho trabajo analiza distintas estrategias para optimizar el consumo y la productividad en situaciones de baja carga.

Capítulo 8

Conclusiones y trabajo futuro

La política de planificación propuesta en este trabajo da soporte de calidad de servicio en arquitecturas CMP. Para obtener resultados experimentales sobre un sistema operativo real, se ha optado por desarrollar un *planificador simbiótico* que implementa esta política sobre Linux kernel 2.6.21. Aunque esta implementación constituye una primera aproximación sobre una arquitectura CMP de dos cores (Intel Core 2 Duo), es posible adaptar dicha implementación a arquitecturas compuestas por más de dos cores; quedando como trabajo futuro la realización de un estudio de las prestaciones de la política de QoS en estos sistemas.

Los resultados obtenidos muestran mejoras significativas del rendimiento de procesos prioritarios en un entorno de ejecución con procesos de distinta prioridad. De manera adicional, la productividad del sistema no se ve degradada al desplegar la política de calidad de servicio; llegando en ocasiones a aumentar la productividad gracias a la disminución del impacto de las situaciones de conflicto por compartición de recursos entre los cores. Por otra parte, como muestran los resultados, la asignación de distintos valores a los parámetros de configuración del *planificador simbiótico* permite ajustar el grado de calidad de servicio y productividad deseados.

Cabe destacar que la herramienta de monitorización del rendimiento incluida en el *planificador simbiótico* –que hace uso de los contadores hardware de la máquina– no sólo proporciona al planificador información sobre los eventos generados por los procesos durante la ejecución, sino que también permite al usuario realizar *profiling* de los procesos proporcionando mecanismos de extracción de la información almacenada en el núcleo sobre las medidas registradas por los contadores hardware. El hecho de estar integrada dentro del planificador de tareas, permite que la herramienta de monitorización sea con-

sciente de los cambios de contexto, realizando la cuenta de eventos hardware de manera independiente por cada proceso.

Como trabajo futuro se propone la exploración de políticas de calidad de servicio que se basan en la combinación de mecanismos de equilibrado de carga basados en el uso de recursos compartidos por cada proceso (*memory-aware*, [30]), con procedimientos de desactivación de cores en situaciones de conflicto. Por otra parte,también es posible emplear la herramienta de monitorización del rendimiento del planificador simbiótico para ampliar la política de ahorro de consumo de Linux sobre multiprocesadores con chips multicore (descrita en [15]) para que opere con independencia de la carga del sistema.

A pesar de que este trabajo se centra en el análisis de la contención por la compartición del segundo nivel de cache en arquitecturas CMP, las ideas aquí expuestas permiten desarrollar políticas de calidad de servicio – basadas en desactivación de procesadores lógicos– sobre arquitecturas SMT (usando el IPC como parámetro de calidad de servicio [27]); y, de manera jerárquica, como combinación de CMP y SMT, a las arquitecturas CMT.

Apéndice A

Contadores hardware de monitorización del rendimiento

A.1. Introducción

Los contadores de monitorización del rendimiento, son registros del procesador que proporcionan los recursos necesarios para contabilizar distintos tipos de eventos que se producen en el chip. Algunos de estos eventos monitorizables son el número de instrucciones retiradas, el número de fallos de cache de último nivel o el número de fallos de predicción de saltos. Para configurar los contadores hardware para la monitorización del rendimiento, así como para obtener los valores almacenados en ellos; el procesador ofrece un subconjunto de instrucciones de la arquitectura reservadas para uso exclusivo del sistema operativo (ensamblador privilegiado). Comúnmente, la lógica del procesador que gestiona los contadores hardware y procesa la configuración de cuenta establecida por el sistema se denomina unidad de monitorización del rendimiento (PMUs, performance monitoring unit).

Los eventos de monitorización son de interés tanto para aquellos que se dedican al análisis y la evaluación del rendimiento de distintas configuraciones hardware, como para desarrolladores de compiladores y de sistemas operativos, elementos que tienen un notable impacto en el rendimiento final de las aplicaciones. Por ejemplo, un analista que observe que un programa presenta una tasa elevada de fallos de cache en un sistema, puede experimentar mejoras en el rendimiento con determinadas opciones de compilación que introduzcan prebúsqueda software o bien que faciliten la prebúsqueda hardware. Por otra, parte un desarrollador de compiladores que observe la misma situación, puede modificar el generador de código específico del backend del

compilador, para que tenga en cuenta los detalles necesarios de la arquitectura subyacente, para optimizar el comportamiento de esa aplicación con la jerarquía de memoria.

El subcomité de SPEC CPU también ha hecho uso de los contadores hardware durante el desarrollo de SPEC CPU 2006 [21]. En este caso, el interés que se persigue no es la mejora del rendimiento para una máquina en particular sino la selección de los candidatos a benchmarks.

Otros usos adicionales de los contadores hardware, incluyen:

- Desarrollo de herramientas de profiling no intrusivas: Son aquellas herramientas de profiling que no requieren la inclusión de código en el binario del programa a evaluar; sino que, por el contrario, permiten obtener información acerca de una aplicación cualquiera -sin procesamientos previos del binario- a partir de las medidas efectuadas por los contadores hardware de la máquina.

- Sintonizaciones del sistema operativo con la arquitectura: A partir de las medidas que el sistema obtiene, en tiempo de ejecución, se definen políticas de planificación para optimizar aspectos como la productividad, el rendimiento o la calidad de servicio.

- Medida del impacto de las nuevas técnicas de alto rendimiento implementadas en los procesadores actuales como la calidad de la predicción de saltos, el impacto de la prebúsqueda hardware o la contención por la compartición de recursos como el segundo nivel de cache. Motivados por el análisis por la contención por la compartición de recursos, arquitecturas como la SMT (multithreading simultáneo) o CMP (arquitecturas multi-core) han sido objeto de múltiples estudios por la presencia de recursos compartidos por los procesadores lógicos y cores existentes en ellas. Este trabajo se encuentra entre uno de esos estudios.

A.2. Consideraciones sobre el diseño

Esencialmente, la unidad de monitorización del rendimiento (PMU) está formada por dos elementos: la lógica de detección de eventos y los contadores hardware. La lógica de detección de eventos contabiliza el número de eventos de un determinado tipo (o conjunto de eventos) que han ocurrido durante un ciclo de reloj. Cada contador hardware recibe como entrada, cada ciclo de reloj, un número de m bits que representa el número de eventos transcurri-

dos durante ese ciclo. Cada ciclo, el contador acumula este valor al que tiene almacenado.

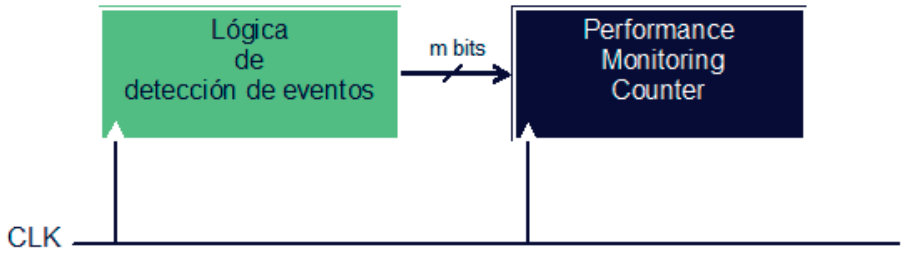

Figura A.1: Estructura de la PMU

Sin embargo, a pesar de la simplicidad de este modelo, la lógica de detección de eventos ha de tener en cuenta las técnicas de alto rendimiento, implementadas en los procesadores actuales, que hacen más complejo [37] el proceso de detección y cuenta de eventos:

- Ejecución en desorden: La ejecución en desorden provoca que las instrucciones ejecutadas fuera de orden, también originen eventos en desorden. Los eventos, como las instrucciones, no se producen según el orden impuesto por el programa.
- Ejecución especulativa: Las instrucciones ejecutadas de forma especulativa, a parte de ser susceptibles de generar falsas excepciones, generan eventos que hacen incrementar las cuentas almacenadas por los contadores hardware. Es preciso disponer de mecanismos para filtrar eventos especulativos y no especulativos.
- Eventos no asociados a instrucciones: Las peticiones de acceso a memoria desencadenadas por las unidades de prebúsqueda hardware, generan eventos no asociados con ninguna instrucción de manera directa.

Las situaciones anteriores hacen necesaria que la lógica de detección y cuenta de eventos, sea capaz de discernir la naturaleza especulativa y fuera de orden de la ocurrencia de eventos. Para hacer frente a este modelo, Intel distingue varios tipos de eventos:

- non-retirement: Son los eventos que suceden durante la ejecución de la instrucción y se contabilizan en el mismo momento en el que suceden, como transacciones en el bus o con la cache. Usar este tipo de eventos implica ignorar la ejecución especulativa y fuera de orden, a la hora de contabilizar los eventos.

- at-retirement: Son los eventos preparados para ser contabilizados cuando se retira la instrucción del ROB –buffer de reordenamiento–, lo que permite contabilizar en orden los eventos efectivos generados por el programa. El mecanismo de conteo de eventos at-retirement implica la inclusión de lógica adicional para etiquetar microoperaciones (*tagging*).

El mecanismo de etiquetado o tagging, empleado para contabilizar eventos at- retirement, separa la ocurrencia de el evento del momento en el que es contabilizado. Cuando el evento se produce, la microoperación generadora se marca en el ROB (tag). Cuando la microinstrucción marcada se retira el evento se contabiliza. La principal limitación de este esquema es que solo puede contabilizarse una sola ocurrencia de un evento at-retirement por microinstrucción. A su vez los eventos at-retirement se dividen en:

- Bogus: Eventos generados por instrucciones ejecutadas especulativamente que pertenecen a la rama de la bifurcación donde se ha producido un fallo en la predicción
- Non-Bogus: Eventos generados por Instrucciones ejecutadas de manera especulativa pertenecientes a la rama de ejecución correcta
- Retired: Eventos generados por instrucciones no ejecutadas de manera especulativa.

Esta clasificación está ligada a la ejecución especulativa. La monitorización de los eventos bogus y non-bogus, es muy útil para estudiar como se comportan las distintas estrategias de predicción de saltos. Gracias a la existencia de eventos de este tipo, se puede conocer el impacto negativo de los fallos de predicción de saltos, ya que permiten contabilizar el número de instrucciones que han sido ejecutadas especulativamente y que, finalmente, desembocaron en un fallo de predicción.

Por otra parte, existen eventos cuya monitorización puede facilitar la detección de situaciones críticas; como la detección de un impacto negativo en el rendimiento provocado por la prebúsqueda hardware. Esta información puede resultar muy valiosa para tomar decisiones que permitan ajustar la máquina a los requisitos de las aplicaciones –en el ejemplo anterior, la prebúsqueda hardware podría deshabilitarse para no entrar en conflicto con un escasa localidad espacial que presenta la aplicación–. Para detectar situaciones de este tipo, en ocasiones, es necesario disponer de un control más preciso que el que pueda ofrecer la simple cuenta de eventos. Por esta razón, la mayor parte de los procesadores actuales incluyen un modelo de cuenta de eventos

precisos (PEBS, Precise Event Based Sampling). Este modelo permite que un subconjunto de eventos at-retirement puedan ser configurados para su detección en modo preciso. En modo preciso, cuando la cuenta de ocurrencias de ese evento at- retirement supera un determinado umbral, se realizan las siguientes acciones:

- Se salva el contexto arquitectónico
- Se lanza una interrupción en el procesador donde se detectó dicha situación
- Se invoca la rutina de tratamiento de interrupción para que realice las acciones necesarias

A.3. Contadores de monitorización del rendimiento en los procesadores de Intel

A.3.1. Interfaz de sistema: MSRs

Los MSRs (Machine/Model Specific Registers) están presentes en los procesadores de Intel desde la aparición del Pentium. Permiten el control de diversas características hardware/software como:

- Contadores de Monitorización del Rendimiento (PMCs)
- Extensiones de depuración
- Mecanismo de detección y notificación de errores del hardware (Machine-Check Architecture)
- Gestión del tipo de *caching* de bloques de memoria (Memory Type Range Registers - MTRs)
- Gestión de consumo y de temperatura (ACPI)

Un MSR es un registro de 64 bits, identificado por un número hexadecimal. Para realizar operaciones de lectura/escritura sobre ellos Intel proporciona las instrucciones: WRMSR y RDMSR. Estas instrucciones estan reservadas para uso exclusivo del sistema operativo (nivel de privilegio 0). Cabe destacar que estas instrucciones hacen uso de operandos implícitos -registros de 32 bits, eax, ecx y edx-. Gracias a su semántica, las aplicaciones que hacen

uso de estas instrucciones no han de sufrir modificaciones en el código fuente para mantener su compatibilidad, pues la instrucción se comporta de manera similar en procesadores pertenecientes a las microarquitectura IA-32 e Intel 64. La semántica de las instrucciones WRMSR y RDMSR se muestra en la tabla A.1.

Semántica de WRMSR	Semántica de RDMSR
`MSR[ecx][0:31] <== eax;`	`eax <== MSR[ecx][0:31]`
`MSR[ecx][63:32] <== edx;`	`edx <==MSR[ecx][63:32]`

Cuadro A.1: Semántica de operaciones de acceso a MSRs

Pueden distinguirse dos tipos de registros MSR: los arquitectónicos y los no arquitectónicos. Los MSRs arquitectónicos, en el momento que se declaran como tales, permenecen inalterables en cuanto a identificador de MSR y funcionalidad. Por otra, parte los MSRs no arquitectónicos pueden sufrir modificaciones en distintos procesadores de la familia, haciendo honor a su nombre de registros específicos de modelo.

Para llevar a cabo la cuenta de un sólo evento, es necesario tener en cuenta más de un registro MSR. Uno de los MSRs necesarios es el PMC (Performance Monitoring Counter), registro que almacena el número de eventos contabilizados de un cierto tipo. Los PMCs son MSRs especiales de 40 bits identificados con un número hexadecimal - distinto del identificador del MSR asociado-. Para acceder a ellos, además de las instrucciones WRMSR y RDMSR, Intel proporciona la instrucción RDPMC. Esta instrucción es la particularización de RDMSR para registros de 40 bits. El identificador numérico de PMC es necesario para acceder a él por medio de la instrucción RDPMC.

Cabe destacar que para que la la lógica de detección y cuenta de eventos incremente correctamente el valor del PMC asignado cada ciclo de reloj, es necesario configurar ciertos aspectos como: el modo de cuenta, la lógica de control del contador, y la selección del evento a monitorizar.

En los procesadores de Intel, existen varios detectores de eventos, cada uno de ellos ubicado en un lugar estratégico del chip para monitorizar un subconjunto determinado de eventos. Los detectores de eventos permiten ser configurados para detectar más de un evento pero no de forma simultánea. Para determinar, que evento contabilizará la unidad de detección, es preciso establecer la configuración en ciertos registros MSRs destinados para tal fin.

Por otra parte, el modo en el que se incrementa el PMC cada ciclo presenta varios modos de funcionamiento, conocidos como modos de cuenta:

- Habilitación de cuenta por nivel de privilegio actual (SO, Usuario): Este modo de cuenta permite habilitar o deshabilitar el conteo de eventos que se producen cuando el proceso esta ejecutando código en modo usuario (binario y bibliotecas, en nivel 3), o bien cuando el procesos ejecuta código del sistema operativo (llamadas al sistema o planificación, en nivel 0).

- Incremento por comparación con umbral: Esta característica resulta muy útil para medir eventos específicos como el número de ciclos de reloj en el que se ha detectado un evento, un número de veces superior o, inferior a un umbral k dado. El contador solo se incrementa cuando se cumple la comparación con dicho umbral. En los registros MSR que permiten configurar el modo de cuenta, existen tres campos reservados: campo de umbral (threshold), campo del signo de la comparación y campo de activación de la característica (edge enable).

- Interrupción por overflow del contador o por incremento del contador: Es el fundamento para los sistemas de monitorización guiados por eventos o guiados por tiempo (sección A.4). La PMU puede configurarse para lanzar interrupciones, en el procesador asociado a un contador determinado, cuando dicho contador se ha desbordado o se haya producido un incremento del mismo superior o inferior a un umbral configurable.

Finalmente, la lógica de control de los PMCs es la que controla la activación y desactivación de la cuenta e indica las situaciones en las que se ha producido un desbordamiento del contador. Para controlar ambas situaciones, existen dos campos reservados (enable y overflow) en el registro MSRs dedicado al control de los contadores.

A.3.2. PMCs de la microarquitectura Netburst

Los procesadores de la microarquitectura Netburst constan de un número variable de MSRs –según el modelo– destinados a la monitorización de eventos del procesador. Para proceder a la configuración de la unidad de monitorización del rendimiento (PMU), es necesario considerar dos tipos de registros MSRs adicionales además de los PMCs: ESCRs y CCCRs. La función de los ESCRs (Event Select Control Register) es permitir la selección del evento que se desea monitorizar en un detector de eventos. Cada detector de eventos tiene asociado un registro ESCR formado por los siguientes campos:

- Flag USR (bit 2). Cuando está activado, los eventos se cuentan cuando el procesador está en el nivel de privilegios (CPL) 1, 2 o 3. Estos niveles son utilizados en las aplicaciones de usuario.
- Flag OS (bit 3). Cuando está activado se cuentan los eventos que se producen cuando el nivel CPL es 0. Este nivel de privilegios, normalmente está reservado al sistema operativo. Cuando este flag y el anterior están activados, se contarán los eventos que se producen en todos los niveles de privilegio.
- Flag Enable (bit 4). Activado cuando se permite el etiquetado de *µops* para contar un eventos at-retirement. Cuando está a 0, el etiquetado está desactivado.
- Tag Value field o Valor de la Etiqueta (bit 5 al 8). Como su propio nombre indica, selecciona un valor para la etiqueta utilizada.
- Event Mask field o Máscara de Evento (bit 9 al 24). Esta máscara es la que permite seleccionar que evento contar de la clase de eventos seleccionada en el campo de selección de evento.
- Event Select field o Selección de Evento (bit 25 al 30). Selecciona la clase de evento. Para seleccionar un evento de esta clase se pondrá el valor correcto en el flag de Máscara de Evento.

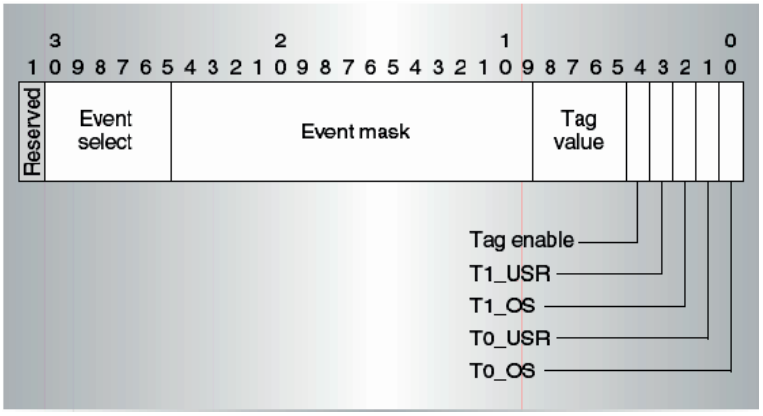

Figura A.2: Campos de un registro ESCR

Por otra parte, los registros CCCR (Counter Configuration Control Register) son registros MSRs, destinados al control del contador y a la configuración del modo de cuenta del mismo. Los registros CCCRs constan de los siguientes campos:

- Flag de activación (Enable flag) (bit 12). Cuando está seleccionado, la cuenta permanece activa. Cuando se borra o se hace un reset del CCCR completo, se pone a 0 y deja de contar.
- Selección de ESCR (ESCR select field) (bits 13 al 15). Identifica el ESCR que se usará para seleccionar los eventos que se contarán en el contador asociado al CCCR.
- Flag de comparación (Compare flag) (bit 18). Cuando está seleccionado se activa el filtro para el contador. Se puede filtrar el valor del contador mediante un umbral, complemento y edge flags.
- Flag de complemento (Complement flag) (bit 19). Indica como comparar el valor del contador con el valor umbral. Cuando está seleccionado, los valores menores o iguales al umbral habilitan el incremento del contador. En caso de que no esté activado, son los valores mayores los que habilitan el incremento.
- Umbral (Threshold field) (bits 20 al 23). Campo para establecer un valor umbral con el que comparar en los filtrados de eventos.
- Edge flag (bit 24). Cuando esta seleccionado activa la salida de la comparación. Sólo funciona cuando está activado el flag de comparación.
- FORCE OVF flag (bit 25). Fuerza a que la señal de desbordamiento se active en cada incremento del contador.
- OVF PMI flag (bit 26). Si está activado entonces se lanzar una interrupción de contador de rendimiento o PMI (performance monitoring-interrupt) cada vez que suceda un desbordamiento en el contador. Si está desactivado no se lanza PMIs.
- Cascade flag (bit 30). Con que este flag esté activado en uno de los contadores que forma la pareja, hace que los contadores funcionen en cascada, es decir que se alternen con otros cuando se produce desbordamiento.
- OVF flag (bit 31). Estará activado cuando el contador haya sufrido un desbordamiento.

En el caso del procesador Intel Pentium 4 Xeon con tecnología hyperthreading, perteneciente a la familia de procesadores de la microarquitectura Netburst, el usuario dispone de:

- 18 contadores de 40 bits (PMCs)
- 149 eventos configurables
- 18 PMCs o contadores
- 18 CCCRs, para la configuración del modo cuenta y control de cada PMC
- 44 ESCRs, para configuración de la lógica de detección de eventos.

Cabe destacar que en el procesador anteriormente citado -procesador con multithreading simultáneo (SMT) de dos vías-, los registros MSRs dedicados a la monitorización del rendimiento están compartidos entre los dos procesadores lógicos

Para configurar la cuenta de un evento es preciso seleccionar un detector de eventos y configurar su registro ESCR asociado. Posteriormente, se ha de elegir un contador hardware (PMC) interconectado con el detector de eventos (según indica la figura), y establecer el valor adecuado de los campos del registro de control y configuración CCCR asociado al contador. Debido a este rígido esquema, el repertorio de eventos no es ortogonal ya que no todos los contadores pueden contar todos los eventos. La especificación de la configuración de los ESCRs se detalla en [38] y [39].

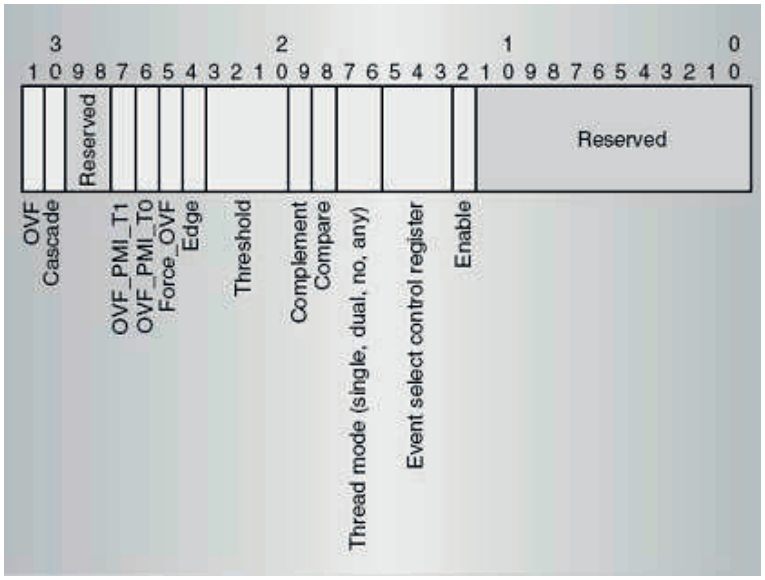

Figura A.3: Campos de un registro CCCR

El procedimiento de configuración de cuenta de un evento comentado anteriormente, es el empleado para eventos non-retirement y en el hay 3 MSRs involucrados. Sin embargo, para llevar a cabo la cuenta de un evento at-retirement, es preciso realizar la configuración equivalente a dos eventos, para levar a cabo el procedimiento de etiquetado (tagging) de forma explícita. El primer evento, representa la acción de detección del evento seleccionado propiamente dicha. Cuando el evento se detecta, la instrucción se marca en la entrada que ocupa en el ROB. El segundo evento involucrado, procede a la detección de las instrucciones etiquetadas -durante la fase de commit- y procede al incremento efectivo del PMC asociado. Cabe destacar que para la configuración de un sólo evento at-retirement, es necesario reservar 6 MSRs-3 para el etiquetado y 3 para el marcado-, frente a los sólo 3 MSRs de los eventos non-retirement.

Las principales limitaciones que presenta la implementación de la PMU en los procesadores de la microarquitectura Netburst son las siguientes:

- El Repertorio de eventos no es *ortogonal*. No todos los contadores pueden contar todos los eventos debido al interconexionado.
- La cuenta de eventos no siempre puede realizarse de manera indepen-

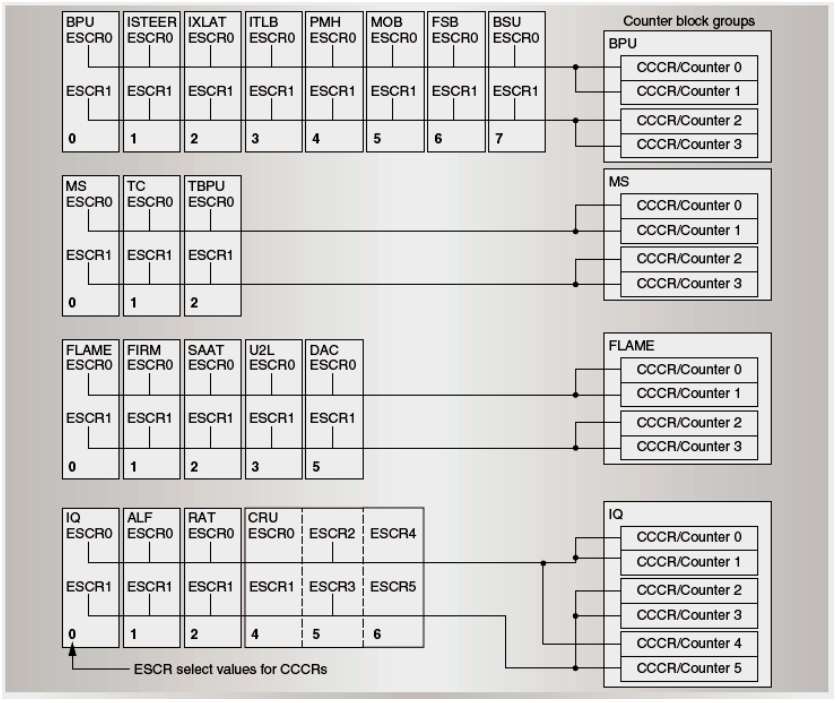

Figura A.4: Interconexionado PMU P4

diente en cada procesador lógico de un procesador con hyperthreading. Intel distingue, por esta situación, entre dos tipos de eventos:

- Eventos Thread Specific (TS): Pueden monitorizarse de manera independiente en cada cpu lógica
- Eventos Thread Independent (TI): La detección se produce sin distinción entre el procesador que emitió la instrucción generadora del evento

A.3.3. PMCs de la microarquitectura Core

Los procesadores de la microarquitectura Core constan de un número variable de MSRs –según el modelo– destinados a la monitorización de eventos del procesador. Estos registros MSR están replicados en cada core de procesador. A diferencia de los procesadores de la microarquitectura Netburst, los procesadores incluyen un subconjunto mínimo de eventos y MSRs arquitectónicos. Este hecho implica que todos los procesadores de la familia comparten un subconjunto común de eventos y de contadores hardware, que garantiza que las herramientas desarrolladas para facilitar el proceso de monitorización de eventos hardware, puedan ser reutilizables en mayor medida, para efectuar mediciones en distintos procesadores de la familia. Este subconjunto mínimo arquitectónico incluye:

- 2 PMCs para cuenta de eventos configurables y 2 registros MSR de configuración asociados (PEREVTSELs)
- 3 PMCs de cuenta fija para llevar a cabo la cuenta de instrucciones retiradas, ciclos de bus y ciclos del procesador
- Un registro MSR de control de los contadores de cuenta fija
- 3 registros MSRs para regir la lógica de control global de los contadores hardware. Esta lógica permite habilitar o deshabilitar de forma global o centralizada las cuentas de los PMCs, así como consultar las situaciones de desbordamiento de los mismos.
- 7 eventos arquitectónicos.

En el caso del procesador Intel Core 2 Duo perteneciente a la familia de procesadores de la microarquitectura Core, el usuario dispone de 190 eventos monitorización para configurar los dos contadores hardware (PMCs) incluidos en la PMU. Dichos eventos se detallan en el apéndice B de [39].

Para proceder a la configuración de la unidad de monitorización del rendimiento (PMU), es necesario considerar un tipo de registro MSR adicional: PERFEVTSEL. El registro PERFEVTSEL se encarga de almacenar la configuración de tres aspectos:

- Selección de evento
- Modo de cuenta
- Lógica de control del contador hardware o PMC asociado

Los distintos campos de un MSR PERFEVTSEL se muestran en la figura . La descripción de cada uno de ellos es la siguiente:

- Unit mask (UMASK) (bits 8 a 15): Estos bits catalogan la condición que la lógica del evento seleccionado detecta. Los valores válidos para cada unidad de detección de eventos es específica a la misma. Para cada evento arquitectónico, su correspondiente valor de campo UMASK define una condición microarquitectónica específica.
- USR (user mode) flag (bit 16): Especifica que la condición microarquitectónica seleccionada se cuenta solamente cuando el procesador lógico opera en el nivel de privilegio 1, 2 o 3. Este flag puede utilizarse en combinación con el flag OS.
- OS (operating system mode) flag (bit 17): Especifica que la condición microarquitectónica seleccionada se cuenta cuando el procesador lógico opera en el nivel de privilegio 0. Este flag puede utilizarse en combinación con el flag USR.
- E (edge detect) flag (bit 18): Cuando está habilitado, activa la detección de umbral de la condición microarquitectónica seleccionada. El procesador lógico incrementará el contador cuando el valor de cuenta de eventos por ciclo supere (o sea inferior- según configuración del resto de campos) al valor introducido en el campo CMASK.
- PC (pin control) flag (bit 19): Cuando esté habilitado, el procesador lógico activa los pins PMi (performance monitoring interrupt) e incrementa el contador cuando ocurren los eventos de monitorización del rendimiento. Si está desactivado el procesador lógico activa los pins PMi cuando el contador se desborde.

- INT (APIC interrupt enable) flag (bit 20): Cuando se active, el procesador lógico genera una interrupción a través de su APIC local cuando el contador se desborda.
- EN (Enable Counters) Flag (bit 22): Señal de enable del contador hardware. Cuando está activado permite la cuenta de eventos.
- INV (invert) flag (bit 23): Invierte el resultado de la comparación de la máscara del contador cuando está habilitado. De este modo, el contador hardware se incrementa cuando el valor de la unidad de detección de eventos para ese ciclo es inferior al indicado por el campo CMASK.
- Counter mask (CMASK) (bits 24 a 31): Cuando este campo es distinto de cero, un procesador lógico compara el valor de este campo con la cuenta de eventos para ese ciclo. Si la cuenta es mayor o igual que este valor, el contador se incrementa en una unidad; de otro modo el contador no se incrementará.

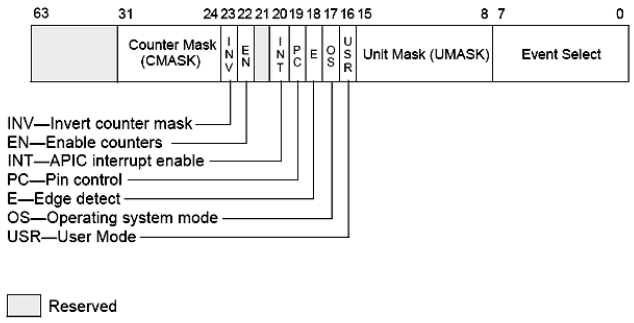

Figura A.5: Campos de un registro PERFEVTSEL

Para configurar la cuenta de un evento es preciso seleccionar un evento y un contador hardware compatible. Para cada contador hardware se debe seleccionar el evento configurable a contabilizar, así como su modo de cuenta. Estos aspectos, unidos al control de activación y desbordamiento del contador, pueden gestionarse configurando adecuadamente el registro PERFEVTSEL asociado a cada PMC. Para poder realizar la monitorización de eventos non-retirement o at-retirement, solamente es necesario seleccionar un evento, un PMC y configurar su registro PERFEVTSEL asociado. Los eventos non-retirement pueden ser contabilizados por cualquier PMC, a diferencia de los eventos at-retirement restringidos al PMC0.

A diferencia del procedimiento empleado para la configuración de los eventos at- retirement en los procesadores de la microarquitectura Netburst; en la microarquitectura Core el procedimiento de tagging, se realiza de forma implícita -sin que la configuración implique la distinción de dos eventos (uno de detección y marcado, y otro de conteo)-. Este hecho, unido a la compactación de las funcionalidades de CCCRs y ESCRs en un sólo registro MSR –PERFEVTSEL–, reduce el número de registros involucrados en la configuración de un evento a sólo dos MSRs.

A.3.4. Comparativa

Con la nueva implementación de los recursos hardware para la monitorización del rendimiento (microarquitectura Core), Intel se ha propuesto mejorar diversos aspectos para facilitar el desarrollo de sistemas que hagan uso de estos recursos. Algunas de las principales diferencias entre ambas implementaciones –microarquitectura Netburst y microarquitectura Core– se muestran en las tablas A.2 y A.3.

A.4. Modelos de sistema

Los recursos de monitorización del rendimiento permiten a investigadores, analistas y desarrolladores de aplicaciones de alto rendimiento, tener acceso a información muy valiosa para llevar a cabo optimizaciones y así conseguir una mayor sintonización de la aplicación o sistema en cuestión con la arquitectura subyacente. Sin embargo, para poder hacer uso de ellos, hay que tener en cuenta diversos aspectos, como la restricción de acceso, los entornos de ejecución multitarea y el bajo nivel expresivo de los eventos de monitorización.

En primer lugar, es necesario considerar que el conjunto de operaciones de acceso a los contadores de monitorización del rendimiento, está reservado exclusivamente para uso del sistema operativo y los controladores de dispositivo (modo núcleo). Por tanto, el sistema operativo o los drivers serán los encargados de ofrecer un interfaz de acceso a los mismos a las herramientas de usuario desarrolladas para simplificar la labor de medición de eventos hardware.

Actualmente la mayor parte de sistemas operativos de propósito general ofrecen un entorno de ejecución multiprogramado -con más de una tarea ejecutando de forma alternada en cada CPU-. Esta situación, unida a la más que

Características	PMCs Core	PMCs Netburst
Número de contadores Hardware disponibles	▪ 18 PMCs (Pentium IV HT)	▪ 2 PMCs de propósito general ▪ 5 PMCs de cuenta fija
Compartición de MSRs entre procesadores lógicos/cores	▪ Compartidos ▪ Configuración para un mismo evento es distinta para cada procesador lógico (MSRs y Valores)	▪ Replicados ▪ Configuración para un mismo evento es similar para todos los cores
MSRs requeridos para la cuenta de un evento	▪ 3 para non-retirement ▪ 6 para at-retirement	▪ 2 MSRs con independencia de la naturaleza del evento
Ortogonalidad del repertorio de eventos	▪ 3 para non-retirement ▪ 6 para at-retirement ▪ Limitada por el interconexionado existente entre los MSRs y las unidades de detección	▪ Todos los PMCs pueden contar eventos non-retirement ▪ Los eventos at-retirement han de se contados con el PMC0
Configuración de cuenta en cascada de PMCs	▪ Soportada ▪ 2 PMCs por evento ▪ (80 bits efectivos para la cuenta de un evento)	▪ No soportada

Cuadro A.2: Comparativa entre PMCs de microarquitectura Core y microarquitectura Netburst (1)

Características	PMCs Core	PMCs Netburst
Complejidad de configuración	▪ Configuración compleja ▪ 3/6 MSRs por evento ▪ Mecanismo de etiquetado (tagging) explícito	▪ Configuración sencilla de PMCs de proposito general (2 MSRs por evento) ▪ Existencia de eventos de cuenta fija (no configurables) ▪ Mecanismo de etiquetado (tagging) implícito
Control centralizado de PMCs	▪ No, solo es posible parar los contadores escribiendo en cada CCCR	▪ Existencia de MSRs de control centralizado. Permite las cuentas de todos los PMCs simultáneamente.
Subconjunto de eventos arquitectónico	▪ Cada conjunto de eventos es específico del modelo, no arquitectónico	▪ Subconjunto arquitectónico de 7 eventos

Cuadro A.3: Comparativa entre PMCs de microarquitectura Core y microarquitectura Netburst (2)

probable imposibilidad de acceso libre al código fuente del sistema operativo, obliga a que el backend –fragmento de más bajo nivel– de las herramientas de monitorización del rendimiento se implemente frecuentemente como un driver del sistema. A pesar de que los drivers pueden tener acceso a los contadores hardware, éstos no disponen de información tan precisa para conocer las situaciones en las que un proceso abandona voluntaria o involuntariamente la CPU para dar paso a otros –cambios de contexto–, como el planificador del sistema operativo. Este hecho dificulta la asociación entre las mediciones de los eventos y los procesos que los generaron.

En último lugar, cabe destacar que los eventos capturados por las unidades de monitorización del rendimiento del procesador (PMUs) son de bajo nivel expresivo; ya que no puede establecerse una correspondencia directa entre un evento hardware y un parámetro de rendimiento -como la tasa de fallos del último nivel de cache o el número de instrucciones por ciclo-. Esta situación obliga a que la obtención de parámetros de rendimiento significativos requiera la monitorización de más de un evento. Por otra parte, el número de eventos que la PMU permite contabilizar de manera simultánea es bastante limitado, lo cual -unido a la baja expresividad de los mismos- dificulta la obtención de informes completos de rendimiento en una sola ejecución.

Estos y otros aspectos han de tenerse muy en cuenta a la hora de desarrollar herramientas de monitorización del rendimiento basadas en contadores hardware. Teniendo en cuenta los distintos procedimientos de monitorización de los eventos hardware que siguen las herramientas actuales, pueden distinguirse tres modelos [40]:

- Sistema basado en intervalo de muestreo o guiado por tiempo
- Sistema guiado por eventos
- Sistema integrado en planificador del sistema operativo

A.4.1. Sistemas basados en intervalo de muestreo

Los sistemas basados en intervalo de muestreo se implementan como drivers del sistema operativo. El objetivo de este tipo de sistemas es el almacenamiento de las cuentas de eventos registradas por los contadores hardware durante intervalos de muestreo fijos. Su procesamiento se activa por tiempo, capturando interrupciones temporizadas en cada CPU. En la rutina de tratamiento de la interrupción realiza el siguiente procesamiento:

1. Almacena los siguientes datos en un buffer interno:

- Contador de programa
- Identificador de la tarea que esta ejecutando en este momento
- Valor almacenado por los contadores hardware (cuenta de eventos)

2. Resetea los contadores y reanuda la cuenta de los eventos
3. . Reprograma la siguiente interrupción temporizada

Las herramientas de usuario procesan la información almacenada en el buffer de datos para asociar las medidas de los contadores con los fragmentos de código ejecutados por los programas monitorizados. Mediante el identificador de la tarea es posible establecer una asociación entre las medidas de los eventos monitorizados, con el fichero binario que almacena el código ejecutado por el proceso que los generó. El contador de programa se utiliza como desplazamiento (ofsset) para acceder al binario. Con esta última información puede asociarse el fragmento de código del programa ejecutado con la información capturada por los contadores hardware; obteniendo -de este modo- información de profiling sin necesidad de modificar el binario de la aplicación para dicho fin.

La ventaja esencial que presenta este tipo de sistemas es que para llevar a cabo su desarrollo, no es preciso que el sistema operativo subyacente sea open-source. Sin embargo, las muestras tomadas entre los cambios de contexto de los procesos de la CPU asociada, han de ser desechadas; ya que no es posible realizar separaciones entre las cuentas que corresponden a cada proceso.

A.4.2. Sistemas guiados por eventos

Los sistemas guiados por eventos también pueden implementarse como drivers del sistema operativo. El objetivo de este tipo de sistemas es la detección de situaciones especiales que pueden ser asociadas con desbordamientos de un contador hardware, o situaciones en las que el número de ocurrencias de eventos de un cierto tipo supere un umbral. Su procesamiento se activa por interrupciones programadas para lanzarse cuando un contador hardware:

1. Se desborda (overflow)
2. El valor almacenado supera un umbral configurable (threshold)

En la rutina de tratamiento de la interrupción realiza el siguiente procesamiento:

1. Captura los siguientes datos:
   - Contador de programa
   - Identificador de la tarea que esta ejecutando en este momento
   - Valor almacenado por los contadores hardware (cuenta de eventos)
2. Resetea los contadores y reanuda la cuenta de los eventos

A diferencia de los sistemas guiados por tiempo, los intervalos transcurridos entre reseteo y reseteo de los contadores hardware no son regulares, sino que dependen de la distribución del número de eventos de un cierto tipo que transcurren a lo largo del programa. A su vez este tipo de sistemas pueden funcionar en modo preciso (PEBS) o impreciso.

Las principales diferencias en cuanto a funcionalidad y prestaciones que presentan este tipo de sistemas frente a los guiados por tiempo, son las siguientes:

- Implican menos sobrecarga que el guiado por tiempo
- Resultan muy eficaces para detectar los *puntos calientes* en la ejecución (e.g fallos de cache)
- Permiten realizar profiling de grano grueso

A.4.3. Sistemas integrados en planificador del sistema operativo

Este tipo de sistemas se integran dentro del planificador de tareas del sistema operativo. Este modelo de sistema permite desarrollar tanto herramientas de monitorización de grano fino como sintonizaciones del sistema operativo en una arquitectura destino concreta. El hecho de estar integrado dentro del planificador de tareas, le permite una total precisión a la hora de asociar las cuentas de eventos de monitorización con las tareas que provocaron dichos eventos. Su procesamiento de captura de la información almacenada en los contadores hardware puede activarse cada n ticks de planificador (intervalos regulares), y/o cada cambio de contexto (intervalos irregulares).

La principal ventaja de este modelo, es la posibilidad de efectuar decisiones de planificación en función de las muestras capturadas. Con ello, pueden diseñarse políticas de planificación que optimicen aspectos como el rendimiento, la productividad o la calidad de servicio en función del comportamiento del sistema que revela la monitorización de los eventos hardware.

Las principales diferencias en cuanto a funcionalidad y prestaciones que presentan este tipo de sistemas al resto de modelos, son las siguientes:

- Implican menos sobrecarga que el guiado por tiempo
- Tienen control absoluto sobre los cambios de contexto[1]
- Permiten el desarrollo de optimizaciones del SO en función de las muestras
- Su desarrollo implica disponer del código fuente del sistema operativo (Linux, FreeBSD, OpenSolaris)

A.5. Dificultades de uso de los contadores hardware

Aunque los contadores hardware proporcionan información útil, para llegar a ciertas conclusiones u obtener mejoras en el rendimiento; esta información puede llegar a ser difícil de interpretar por diversas causas:

- Perspectiva: En ocasiones, los contadores registran eventos que son más útiles para los arquitectos hardware que para los investigadores o desarrolladores. Por ejemplo, es bastante infrecuente que los contadores hardware diferencien entre los eventos que registran el número de operaciones ejecutadas por una unidad funcional de forma voluntaria (siguiendo el orden de las instrucciones del programa) o involuntaria (ejecución especulativa y planificación dinámica).
- Existencia de eventos innecesarios: Las unidades de monitorización del rendimiento (PMUs, performance monitoring units) incluidas en los procesadores actuales soportan la cuenta de más eventos de los necesarios para una analista para llevar a cabo sus pruebas. En ningún caso, estos eventos extra podrían ayudar a encontrar un cuello de botella en una aplicación.

[1]Cabe destacar que los sistemas de monitorización integrados en el planificador de tareas tienen control sobre los cambios de contexto de los threads en ejecución. En los otros modelos, un cambio de contexto entre interrupción e interrupción fuerza a desechar la medida realizada ya que se contabilizan valores de más de un proceso.

- Multiplexación: Aunque un chip pueda implementar muchos eventos de rendimiento, solo es posible contar un pequeño número de ellos al mismo tiempo en la mayoría de las implementaciones. Las soluciones adoptadas, por investigadores y analistas para afrontar este problema, no consisten en realizar tantas ejecuciones de un programa como subconjuntos de eventos se desee contablizar, sino alternar durante la ejecución –de forma rotativa– la cuenta de cada uno de estos subconjuntos. Esta técnica se conoce como multiplexación.

- Escaso presupuesto para testeo: Los contadores de monitorización de rendimiento y su interfaz software- subconjunto de ensamblador priviliegiado accesible a nivel de sistema operativo- no son usadas tan extensivamente como otras características del procesador, y por lo tanto se destina un presupuesto inferior para testeo. Por ejemplo, en una implementación de una PMU se detectó que dicha unidad fallaba si un proceso era trasladado de una CPU a otra provocando un desbordamiento de un contador de 32 bits. Los usuarios de esta PMU, debían ser conscientes de ese problema de implementación y desarrollar soluciones alternativas para poder realizar medidas fiables.

- Importancia del compilador: Si se está llevando a cabo la cuenta del porcentaje de operaciones de punto flotante por segundo, dicha medida esta considerablemente afectada por el impacto en el código de la acción del compilador. Intuitivamente, el porcentaje de operaciones de punto flotante por segundo crece con el nivel de optimización ya que la planificación estática de instrucciones realizada por el compilador puede disminuir el número de paradas del pipeline.

- Confidencialidad y calidad de la documentación: Debido a que los eventos de rendimiento son tan cercanos a la implementación de cada procesador, los fabricantes tienden a mantener ocultos sus informes completos. En ellos se detalla con exactitud, la semántica de cada evento y el impacto mostrado por cada uno de ellos sobre cada una de las técnicas de alto rendimiento incorporadas en el procesador. Es por ello que dicha información puede permitir sacar conclusiones sobre aspectos de implementación que no han sido desvelados públicamente.

A pesar de estos inconvenientes, los contadores hardware son irremplazables por su gran importancia, ya que gracias a ellos se pueden detectar directamente aquellas características del procesador y del subsistema de memoria cache que constituyen verdaderos cuellos de botella en el rendimiento. El desarrollo de nuevas herramientas como SPOT, Pappy, oprofile,... permiten

realizar la cuenta de eventos de manera más sencilla. Sin embargo, los procesadores tienden a ser más complejos, incluyendo eventos más detallados para monitorizar; dificultando el desarrollo de herramientas que simplifiquen estos pequeños detalles, y haciendo más complejo equiparar los distintos eventos entre arquitecturas.

Bibliografía

[1] J. Lo, S. Eggers, J. Emer, H. Levy, R. Stamm, and D. Tullsen. Converting thread-level parallelism into instruction-level parallelism via simultaneous multithreading. *ACM TOCS 15*, 1997.

[2] J. M. Borkenhagen, R. J Eickemeyer, R.Ñ. Kalla, and S.R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development 44*, 2003.

[3] K. Olukotun, B.Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. *ASPLOS*, 1996.

[4] J. Lo et al. Initial observations of the simultaneous multithreading pentium 4 processor. *PACT*, 2003.

[5] A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. *ISCA*, 1998.

[6] K. Keeton, D. Patterson, R. Raphael Y. He, and W. Baker. Performance characterization of a Quad Pentium Pro SMP using OLTP workloads. *ISCA*, 1998.

[7] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *ISCA*, 1995.

[8] RonaldÑ. Kalla, Balaram Sinharoy, and Joel M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[9] Jonathan Schwartz. Sun′ Niagara processor. *http://blogs.sun.com/roller/page/jonathan/20040910*, 2003.

[10] IBM Corporation. IBM eServer iSeries announcement. *http://www-1.ibm.com/servers/eserver/iseries/announce/*, 2000.

[11] Alexandra Fedorova, Margo I. Seltzer, Christopher Small, and Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX Annual Technical Conference, General Track* [41], pages 395–398.

[12] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS*, pages 234–244, 2000.

[13] Fei Guo, Hari Kannan, Li Zhao, Ramesh Illikkal, Ravi Iyer, Don Newell, Yan Solihin, and Christos Kozyrakis. From chaos to QoS: case studies in CMP resource management. *SIGARCH Comput. Archit. News*, 35(1):21–30, 2007.

[14] Robert Love. *Linux Kernel Development*. Sams Publishing, 2004.

[15] Suresh Siddha, Venkatesh Pallipadi, and Asit Mallick. Chip multi processing (CMP) aware Linux kernel scheduler. *OLS*, 2005.

[16] Intel Corporation. *Intel 64 and IA32 Architectures Software Developer's Manual Volume Optimization Reference Manual Order Number 248966*. Intel Corporation, 2007.

[17] Francisco J. Cazorla, Alex Ramírez, Mateo Valero, Peter M. W. Knijnenburg, Rizos Sakellariou, and Enrique Fernández. QoS for High-Performance SMT processors in embedded systems. *IEEE Micro*, 24(4):24–31, 2004.

[18] James R. Bulpin and Ian Pratt. Hyper-Threading aware process scheduling heuristics. In *USENIX Annual Technical Conference, General Track* [41], pages 399–402.

[19] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Subsetting the SPEC CPU2006 benchmark suite. *SIGARCH Comput. Archit. News*, 35(1):69–76, 2007.

[20] John L. Henning. SPEC CPU2006 memory footprint. *SIGARCH Comput. Archit. News*, 35(1):84–89, 2007.

[21] John L. Henning. Performance counters and development of SPEC CPU2006. *SIGARCH Comput. Archit. News*, 35(1):118–121, 2007.

[22] Frank Bellosa and Martin Steckermeier. The performance limits of locality information usage in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 37(1):113–121, 1996.

[23] Frank Bellosa. Follow-on scheduling: Using TLB information to reduce cache misses. *Symp. on Operating Systems Principles - Work in Progress Session*, 1997.

[24] P. Koka and M. H. Lipasti. Opportunities for cache friendly process scheduling. *Workshop on Interaction Between Operating Systems and Computer Architecture*, 2005.

[25] P. Koka and M. H. Lipasti. Enhancements for hyper-threading technology in the operating system seeking the optimal micro-architectural scheduling. *Workshop on Industrial Experiences with Systems Software*, 2002.

[26] S. Parekh, S. Eggers, H. Levy, and J. Lo. Thread-sensitive scheduling for SMT processors. *Technical report, Dept. of Computer Science and Engineering, Univ. of Washington*, 2000.

[27] Ali El-Moursy, R. Garg, David H. Albonesi, and Sandhya Dwarkadas. Compatible phase co-scheduling on a CMP of multi-threaded processors. In *IPDPS*. IEEE, 2006.

[28] Robert L. McGregor, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *IPDPS*. IEEE Computer Society, 2005.

[29] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. Effects of memory performance on parallel job scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *JSSPP*, volume 2221 of *Lecture Notes in Computer Science*, pages 116–132. Springer, 2001.

[30] G. Edward Suh, Srinivas Devadas, and Larry Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, pages 117–, 2002.

[31] A. Fedorova, C. Small, D.Ñussbaum, and M. Seltzer. Chip multithreading systems need a new operating system scheduler. *SIGOPS European Workshop*, 2004.

[32] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. *SIGMETRICS Perform. Eval. Rev.*, 35(1):25–36, 2007.

[33] Ravi R. Iyer. CQoS: a framework for enabling QoS in shared caches of CMP platforms. In Paul Feautrier, James Goodman, and André Seznec, editors, *ICS*, pages 257–266. ACM, 2004.

[34] Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernández, Alex Ramírez, and Mateo Valero. Implicit vs. explicit resource allocation in SMT processors. In *DSD*, pages 44–51. IEEE Computer Society, 2004.

[35] Francisco J. Cazorla, Alex Ramírez, Mateo Valero, and Enrique Fernández. Dynamically controlled resource allocation in SMT processors. In *MICRO*, pages 171–182. IEEE Computer Society, 2004.

[36] Francisco J. Cazorla, Peter M. W. Knijnenburg, Rizos Sakellariou, Enrique Fernández, Alex Ramírez, and Mateo Valero. Feasibility of QoS for SMT. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 535–540. Springer, 2004.

[37] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Z. Chrysos. *ProfileMe*: Hardware support for instruction-level profiling on out-of-order processors. In *MICRO*, pages 292–302, 1997.

[38] Brinkley Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, 2002.

[39] Intel Corp. *Intel 64 and IA32 Architectures Software Developer's Manual Volume 3A & 3B System Programming Guide, Part 1 & 2, Order Number 253668, 253669*. Intel Corporation, 2007.

[40] Brinkley Sprunt. The basics of performance-monitoring hardware. *IEEE Micro*, 22(4):64–71, 2002.

[41] *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 2005.